本书从区块链的基础知识开始讲起，以HyperLedger Fabric为主线，清晰讲解区块链的原理、源码、搭建与应用，可帮助读者轻松上手区块链。

Broadview®
www.broadview.com.cn

区块链轻松上手
原理、源码、搭建与应用

Leader-us 李艳军 赵锴 编著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

作者简介

· Leader-us ·

本名吴治辉，HPE资深架构师，拥有超过15年的软件研发经验，专注于电信软件和云计算领域的软件研发，拥有丰富的大型项目架构设计经验，是业界少有的具备很强Coding能力的S级资深架构师，也是《ZeroC Ice权威指南》《架构解密：从分布式到微服务》《Kubernetes权威指南：从Docker到Kubernetes实践全接触》《Kubernetes权威指南：企业级容器云实战》等书的作者。

· 李艳军 ·

拥有多年IT行业从业经验，开源软件爱好者，专注于区块链、云计算方面的技术研究。

· 赵锴 ·

拥有十多年IT行业从业经验，热爱开源事业，致力于将前沿技术转化为生产力，曾在多家手游、电信及医疗公司担任架构师和技术总监。

区块链轻松上手
原理、源码、搭建与应用

Leader-us 李艳军 赵锴 编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书首先从以比特币为代表的数字货币的历史与现状开始，讲解区块链的概念、生态、底层技术与架构；然后讲解 Fabric 的开发环境与调试方法，并细致解析配置文件及命令行的用法；其次以 Fabric Java SDK 为主介绍如何使用 Java 代码开发 Fabric 应用，包括客户端管理、通道配置、事件监听、智能合约开发等；再次深入解析 Fabric 源码，解析客户端交易、智能合约初始化及背书流程；最后深入讲解 Fabric 的安全机制，以及 Fabric CA 的使用与管理等内容。

本书兼顾原理与实战，主要面向想快速上手区块链及了解其原理与架构的学生、爱好者、开发人员、架构师与技术管理人员。

图书在版编目（CIP）数据

区块链轻松上手：原理、源码、搭建与应用 / Leader-us 等编著. —北京：电子工业出版社，2018.10
ISBN 978-7-121-34878-5

Ⅰ. ①区… Ⅱ. ①L… Ⅲ. ①程序设计 Ⅳ. ①TP311.1

中国版本图书馆 CIP 数据核字（2018）第 184611 号

策划编辑：张国霞
责任编辑：孙学瑛
印　　刷：三河市双峰印刷装订有限公司
装　　订：三河市双峰印刷装订有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787×980　1/16　印张：15.75　字数：350 千字
版　　次：2018 年 10 月第 1 版
印　　次：2018 年 10 月第 1 次印刷
印　　数：3000 册　　定价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

前　言

在说什么是区块链之前，先说一个小故事。

看过《三国演义》的人都知道，刘备自称刘皇叔，是中山靖王之后，以匡扶汉室之名，拉拢一批人建立了蜀国，形成三国鼎立之势。但回过头来看，大家为什么相信他真的是“刘皇叔”呢？其实在《三国演义》里有这么一段描述：

帝宣上殿，问曰：“卿祖何人？”玄德奏曰：“臣乃中山靖王之后，孝景皇帝阁下玄孙，刘雄之孙，刘弘之子也。”帝教取宗族世谱检看，令宗正卿宣读曰：“孝景皇帝生十四子。第七子乃中山靖王刘胜。胜生陆城亭侯刘贞。贞生沛侯刘昂。昂生漳侯刘禄。禄生沂水侯刘恋。恋生钦阳侯刘英。英生安国侯刘建。建生广陵侯刘哀。哀生胶水侯刘宪。宪生祖邑侯刘舒。舒生祁阳侯刘谊。谊生原泽侯刘必。必生颍川侯刘达。达生丰灵侯刘不疑。不疑生济川侯刘惠。惠生东郡范令刘雄。雄生刘弘。弘不仕。刘备乃刘弘之子也。”帝排世谱，则玄德乃帝之叔也。帝大喜，请入偏殿叙叔侄之礼……

原来就是翻出族谱，追溯整整十八代，才相信刘备为汉室之后。事实上，社会因为“信任”问题需要付出极大的代价，而解决该问题的方法之一就是从可以追溯且不能修改的记录中找到信任的依据。这种信任的实现方式就是讨论区块链的基础。

区块链到底是什么？比特币为什么这么值钱？那些看不见也摸不着的数字货币到底是不是传销？毫无疑问，作为区块链技术的应用之一——比特币已经大获成功，区块链所涉及的账本、分布式与去中心化、共识算法、智能合约、数字密钥、隐私保护、可信计算等技术也变得非常热门，基于这些技术的大量项目涌现。而区块链的发展价值就在于试图通过技术手段降低社会信任成本，并提高社会生产效率。

当然，区块链现在还有不足之处：除了比特币，还没有特别成功的典型应用。究其原因，一方面是区块链在高并发、低延迟的交易场景下还有许多技术问题需要解决；另一方

面是只能保证线上数据可信的特性限制了其应用场景。在大规模应用区块链时，社会的接受成本也是我们必须考虑的要素。在商业利益的驱动下，即使区块链能够提供各种各样的好处，选择应用区块链也只是一种纳什均衡而非最优策略。无论如何，区块链并不是“包治百病的灵丹妙药”，它还只是一个崭新的领域，正在蓬勃发展。

本书总计 6 章：第 1 章从以比特币为代表的数字货币的历史与现状开始，讲解区块链的概念，并通过一个简单示例让读者与 Fabric 有一次“亲密接触”；第 2 章阐述区块链的生态、底层技术与架构；第 3 章讲解 Fabric 的开发环境与调试方法，介绍更复杂的 Fabric 网络，并细致解析配置文件及命令行的用法；第 4 章以 Fabric Java SDK 为主介绍如何使用 Java 开发 Fabric 应用，包括客户端管理、通道配置、事件监听、智能合约开发等；第 5 章从创世区块开始，逐步深入解析 Fabric 源码，解析客户端交易、智能合约初始化及背书流程；第 6 章深入讲解 Fabric 的安全机制，以及 Fabric CA 的使用与管理。本书提供了部分示例代码(参见 GitHub 网站的 MyCATApache/SuperLedger 项目),希望对读者有所帮助，也希望读者能及时反馈并与我们沟通，指出书中的错漏之处，帮助我们完善内容。

最后，感谢家人的理解与支持，感谢张国霞编辑的耐心指导，感谢 Mycat 社区的帮助与鼓励！

目 录

第 1 章
全面理解区块链

1.1 从比特币开始

1.1.1 颠覆性的比特币

比特币（BTC）于 2008 年被中本聪首次提出，其价格在三年后首次突破 1 美元，此后一路飞涨并引发全球投资者的关注。2017 年 2 月 8 日，一枚比特币的价格超过 1 盎司（约 28.35 克）黄金的价格；2017 年 8 月，比特币之父中本聪“高位套现”3000 枚比特币；2017 年 11 月至 12 月初，一枚比特币的价格达到迄今为止的最高点，接近两万美元。

比特币并不是真实的货币，而是一种数字化的货币（简称数字货币）。与黄金一样，比特币的发行方式、发行总量及发行速度并不是由某个国家的央行以不透明方式控制的。可一句话概括：比特币是一种发行透明的、去中心化的、自动控制的数字货币。

首先，比特币的发行规则是完全透明的。比特币不像任一国家的纸币，每年印多少钞票完全由这个国家的央行人为决定。比特币的设计者为了避免比特币的发行数量过多，导致类似纸币“通货膨胀”现象的发生，在设计之初就确定了比特币总量有限的规则：比特币的总量为 2100 万个，在 2009 年生成第一批 50 枚比特币，此时的货币总量就是 50，之后新币以约每 10 分钟 50 个的速度发行；当未发行的比特币数量减少到总量的 50%

即 1050 万时，新币的发行速度就开始减半，约每 10 分钟产生 25 个新币，依此类推。当未发行的比特币数量减少到总量的 25%即 525 万时，新币的发行速度继续减半，并基本上保持新币发行量每 4 年减半的节奏，预计在 2140 年以后不再产生新币。因此，比特币基本能保持零通胀的趋势，不像大部分纸币，随着时间的流逝，其购买力和价值不断缩水。

其次，比特币是一种去中心化的货币。比特币平台基本上是“零门槛”入门的，门槛低到什么程度？只要有一台可以连接互联网的计算机，再加入比特币的 P2P 分布式网络中，就能参与比特币的发行和交易等核心业务了。在比特币平台上，任何个人或者机构都可以自由参与，不需要任何机构的审批，在比特币网络中也不存在任何中心控制节点，每个加入比特币网络中的运算节点都是平等的，它们之间的差别仅仅是算力的多少，每个节点都“努力工作”，用算力来证明自己，这或多或少地影响着比特币的发行、交易、记账等核心业务。

最后，比特币是一种自动控制的数字货币。比特币是一套自动化的软件代码，新币的发行、身份的验证、交易的确认、账本的记录等流程都是由比特币网络中的计算机节点（矿工，或称 Miner）自动执行的，全程没有任何人工参与，需要终端用户（User）参与的只有比特币的买卖操作。每个用户都拥有一个比特币账号，这个账号由一对密钥及比特币的钱包地址（Bitcoin Addr）组成，比特币的买卖操作就是一个用户把比特币转到另一个用户的钱包地址中，如果一个用户把自己所有的比特币都转卖出去，就可以认为是清仓套现。

1. 深入理解“挖矿”及其原理、机制

中本聪在设计比特币之初就限定了比特币的总量，可以想象这些比特币就好像黄金一样被埋在地下，等待着无数玩家（又称矿工）去挖掘（又称挖矿）。一枚比特币的诞生源自比特币矿工的辛劳挖矿，当然，这里的“挖矿”并非指真实的挖矿山的行为，而是矿工通过专有的机器（又称矿机）计算和挖掘出一个很难得到的随机数，谁先计算出来，谁就得到相应的比特币。如果看过斯皮尔伯格的《头号玩家》这部电影，你就能很快弄明白挖矿是怎么回事了。而无数矿机夜以继日地挖矿，挖掘出一枚枚新币的过程，就是比特币的发行过程。

如果想成为比特币网络中的一名矿工，就得先购买一套不错的装备——矿机。一开始，人们都用自己的计算机去做矿机，因为僧少粥多，所以最早的矿工们还能轻松挖到比特币，但随着比特币市值的一路飙升，大批淘金者蜂拥而入，同时，新币的数量在不断减少，基本上每 4 年就减少一半，这时人们基本上都得拼装备了，谁的装备越强，谁的挖矿速度就越快，矿工们不得不购买算力更强的装备。后来，很多矿工自发形成各种挖矿组织——矿

池，最后，比特币的“矿山”成为一个完全数字化的战场，遍布全球的散户与各种“矿池巨兽”一起角逐，其场面像极了《头号玩家》里的“绿洲”。更有趣的是，《头号玩家》里的 101 组织是一个大玩家，不仅靠售卖游戏装备获利，还派人参与争夺游戏中的彩蛋；在比特币世界中也有这样的大玩家，也就是制造、出售矿机和拥有自己的矿池的玩家，其中的头号玩家是我国的比特币大陆（Bitmain）公司，比特币大陆公司 2017 年赚的钱在同期甚至超过知名的英伟达公司！有分析师认为：比特币大陆公司在 2017 年的运营利润为 30 亿 ~ 40 亿美元，而英伟达同期的运营利润为 30 亿美元，比特币大陆公司在短短 4 年内取得的成绩，英伟达却用了 24 年，这就是新经济的速度吧。

现在，我们不太可能靠挖矿致富了，但可以尽力理解挖矿背后的原理。在前面已经了解到，挖矿就是各个矿机去计算和挖掘一个很难得到的随机数，实际上，这是一个纯粹的、拼蛮力、拼算力的竞赛。挖矿算法也被称为工作量证明（Proof of Work，PoW）算法，即矿机不断地重复一些简单的 SHA256 哈希运算，直到找到符合条件的目标哈希值。举个例子，给出以下游戏规则：

> 给定一个基本的字符串"Hello, world!"，可以在这个字符串后面添加一个叫作 Nonce 的整数值，然后对新的字符串执行 SHA256 哈希运算，如果得到的哈希结果（以 16 进制的形式表示）是以“0000”开头的，就找到了符合要求的一个哈希值，谁先找到这个哈希值并上报对应的 Nonce 值，谁就是赢家，谁就能赢取奖品。

按照游戏规则，假设 Nonce 值是从零开始的，则我们不停地递增 Nonce 值，然后对新拼接的字符串执行 SHA256 哈希计算，所以需要经过 4251 次计算才能找到前 4 位恰好为 0 的哈希散列，部分计算的输出结果如下：

```
   "Hello, world!0" =>
1312af178c253f84028d480a6adc1e25e81caa44c749ec81976192e2ec934c64
   "Hello, world!1" =>
e9afc424b79e4f6ab42d99c81156d3a17228d6e1eef4139be78e948a9332a7d8
   ...
   "Hello, world!4250" =>
0000c3af42fc31103f1fdc0151fa747ff87349a4714df7cc52ea464e12dcd4e9
```

上述例子中的 Nonce 就是我们挖矿所要找的那个随机数！我们通过 SHA256 算法得到的是一个长度为 256 位（bit）的数值，可以把计算这个哈希值的过程类比为一次次“抛硬币”的过程。好比有 256 枚硬币，对每个硬币给定依次为 1、2、3 直到 256 的编号，每进行一次哈希运算，就像抛一次硬币，256 枚硬币要同时抛出，要求编号 1 到 N（$0<N<256$）的所有硬币在落地后全部正面向上！试想一下，如果 N 是 128，则要活到几千岁才有可能

抛出一个符合条件的结果，别无他法。所以，矿机只能进行一次次的重复运算，同时比拼速度，谁能更早地计算出这个 Nonce，谁就是赢家，这就是所谓的工作量证明（PoW）。PoW 的流程如图 1-1 所示。

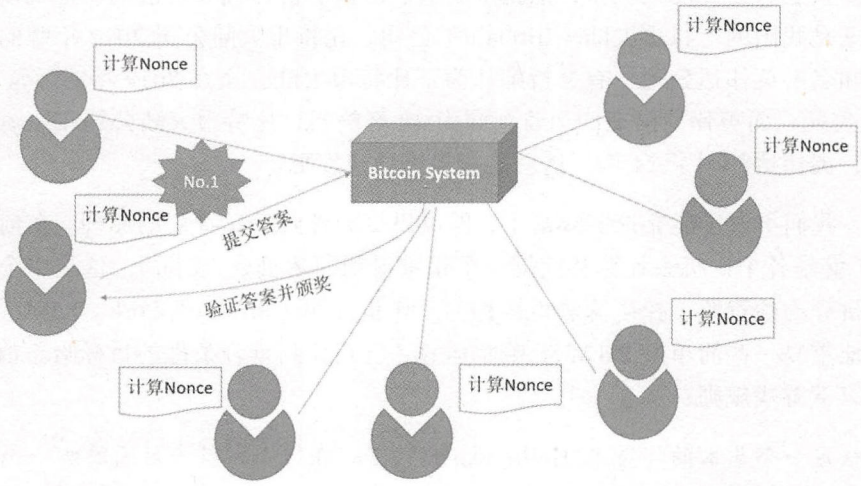


图1-1

在 SHA256 计算中，N 每增大一次，计算目标 Nonce 的工作量就增加很多，矿机不得不花费更多的时间去完成计算，这就减慢了比特币的“造币”速度，因此在挖矿算法中，N 对应的就是 Difficulty（难度值）这个关键参数。Difficulty 会根据全网算力的变化进行自适应调整，以保持造币的速度。中本聪规定每生成 2016 页账本（Block），挖矿的难度就自动增加一级，这就是前面所说的，比特币的新币发行量基本上保持每 4 年减半的节奏的真正原因。虽然寻找这个随机数需要大量的重复运算，但由于这个工作是矿工交给自己的矿机全天 24 小时自动执行的，因此只要装备好，矿工还是有机会“勤劳致富”的。其结果就是全球各地的无数矿工们乐此不疲地“挖矿”，然后在比特币交易市场中出售、变现，再购买更高级的矿机继续挖矿。

由于比特币的总量有限，所以比特币终有一天会被挖完，若要长久地鼓励矿工们为比特币网络“打工”，就需要挖矿以外的激励制度，这就是交易手续费（Transaction Fees），类似于银行转账业务中的手续费。在通常情况下，比特币的转账交易的确是免费的，但在某些情况下必须支付手续费才能完成转账，这是为了激发矿工们的热情，因为矿工打包、转账、交易需要耗费资源（带宽、存储、验证），对矿工们的这些付出进行付费是合理的，同时，针对小额比特币交易则强制收费，这样可以防止黑客用大量的小额比特币交易来冲

击整个 P2P 网络，但收取的费用通常很少，比如 0.000 1 枚比特币。这就带来了另一个值得我们思考的问题，即比特币的总数虽然有限，只有 2600 万枚，但一枚比特币并不是一枚“硬币”，它是可以继续分割的，一枚比特币可以被分割到小数点后 8 位的程度，所以比特币的最小单位是 0.000 000 01BTC，因此，有些人认为比特币仍然是一种可以膨胀的数字货币。

2. 比特币的交易机制

在挖到比特币以后，最重要的就是兑现真实的货币了，毕竟矿工挖矿也是有成本的，特别是电力消耗及矿机长期高负荷运行所带来的损耗。此外，在比特币世界中充斥着大量的比特币投资者，对于这些投资者来说，最重要的就是能够像炒股票一样方便地买入和卖出比特币，以获取差价了。这就涉及比特币交易的问题。我们知道，每个比特币用户都有一个比特币钱包（Bitcoin Wallet），这个钱包有一个接收比特币转账的地址——Bitcoin Address（又称比特币地址），类似于支付宝账号或者银行卡账号，对比特币的交易（Bitcoin Transaction）就是将比特币从一个比特币地址提出，转到另一个比特币地址，在这个过程中不必将比特币转换成人民币等货币，但比特币的受让方必须支付等值资金给比特币的出让方，这就是比特币的场外交易，即一方通过支付宝、微信或银行转账将资金打给另一方。由于这种私下的场外交易存在很高的风险，因此需要一个第三方的交易市场来完成比特币的交易活动。在很长一段时间内，我国拥有全世界最大的交易市场，掌控着世界八成以上的交易量，但从 2017 年下半年起，比特币中国、火币网、OKCoin 币行等几个国内知名的交易市场陆续关停，在我国境内不再存在合法的比特币交易市场。2018 年，国外比较知名的数字货币交易市场有币安、Bittrex、Poloniex 等。

如果认真阅读了上面这段关于比特币交易的说明，你就可能发现一个“疑点”，即在比特币钱包里有一个用来收取比特币的“比特币地址”，却没有账户余额的信息，这完全违背了我们对支付系统的理解！在我们的理解中，所有支付系统都是以账户余额为核心进行设计的，银行也好，证券交易系统也好，互联网第三方支付系统也好，都基于账户余额的设计思想，在数据库里每个账号都有一个余额属性，保存当前账户的资产金额。举例来说，鸣人的钱包里有 100 元，路飞的宝箱里有 50 元，当鸣人要购买路飞捕捞的一只澳洲龙虾交给 Mycat 总部的厨神做定制拉面时，鸣人需要支付 30 元给路飞，其转账交易过程如下。

（1）平台检查鸣人的账户余额是否充足，如果不足 30 元就终止交易，转账失败。

（2）若鸣人的账户余额充足，则在鸣人的账户里减去 30 元（不考虑手续费）。

（3）在路飞的账户里增加 30 元。

（4）交易成功，保存交易流水记录。

但是，在这个看似简单、直观的流程背后，有很多的约束条件及实现代价。

（1）首先，我们需要一个高度可靠的关系型数据库，并且这个数据库严格遵循 ACID 规范。

（2）其次，我们需要编程来确保上述 1 到 3 的过程是在一个数据库事务中执行的，中间任何一步失败，则都需要回滚。

由于事务的约束和限制，我们需要一个中心化的账户系统来保存所有参与交易的账号信息，并且随着交易量的增加，系统中的交易流水记录也会迅速膨胀。试想一下，若全球有超过 100 亿个用户使用比特币，则这个中心化的账户系统的数据库会有多大？因此传统的、中心化的、基于关系数据库的账户余额的设计思路根本无法满足比特币的交易需求，于是中本聪丢弃了中心化的账户数据库的设计思路，独创了以 UTXO 为核心的交易系统，彻底规避了传统交易系统的局限性。UTXO（Unspent Transaction Output，交易输出）的设计受到了业界的高度评价，斯坦福大学密码学与计算机安全教授 Dan Boneh 对 UTXO 的设计给予了很高的评价。

要深入理解 UTXO，最简单的办法就是把一枚比特币从诞生到在市场上流通的整个环节再理解一遍。还是以鸣人买龙虾为例，假设回到故事的起点，鸣人得到蛤蟆仙人的指点，要打败最终的 Boss，需要组织史上最强打怪军团，这需要很多经费，而现在赚取经费的最好方式是挖矿；当然，鸣人天赋异禀，拥有火影多重影分身禁术，天生就是一名挖矿好手！按照当前的市值，鸣人只要挖 100 枚比特币就可以组建一只华丽的打怪军团。于是鸣人一路狂奔去挖矿，路上遇到赶赴好莱坞的黄飞鸿，鸣人看他轻功了得，于是说服他加入打怪军团，一路上又收了远方表弟路飞，三人很快抵达最大的比特币挖矿场地——比特币绿洲，有路飞与黄飞鸿保驾，鸣人施展火影多重影分身禁术，开启并行挖矿，不到 1 小时就挖出了 120 枚比特币，这 120 枚比特币的收入会在比特币的账本里被记录为一笔交易，每笔交易都有若干笔资金来源，即交易输入或者转账人，也都有若干笔资金去向，即交易输出或者收款人，这个交易输出对于收款人来说，就是未花费过的交易输出，即 UTXO。如下所示为鸣人挖矿获取 120 枚比特币的交易记录，这个记录被称为生产交易或者 CoinBase 交易，经常是每个区块的第 1 个交易。

CoinBase 交易

交易编号：#1001

交易输入（Transaction Input）	交易输出（Unspent Transaction Output）		
资金来源	条目编号	金额	收款人地址
挖矿所得	1	120	鸣人

由于打怪军团只需要 100 枚比特币的经费，所以，接下来鸣人分给路飞和黄飞鸿各 10 枚比特币，于是分别产生了两笔交易，这两笔交易的内容分别如下。

普通交易

交易编号：#2001

交易输入（Transaction Input）	交易输出（Unspent Transaction Output）		
资金来源	条目编号	金额	收款人地址
#1001-1	1	10	路飞
#1001-1	2	110	鸣人

普通交易

交易编号：#3001

交易输入（Transaction Input）	交易输出（Unspent Transaction Output）		
资金来源	条目编号	金额	收款人地址
#2001-2	1	10	黄飞鸿
#2001-2	2	100	鸣人

每笔交易都需要做到交易输入与交易输出平衡，因此，在编号为#2001 的鸣人给路飞转账的交易中虽然只有一次转账，但也产生了两条记录。实际上，我们可以将 UTXO 理解为转账流水记录+余额的复式结构，每笔交易的 UTXO 必然是下一笔交易的输入项，在一系列交易后，在鸣人的账号上（比特币地址）最终登记了 100 枚未花费的比特币，路飞与黄飞鸿则各有 10 枚，此外，以区块链方式记录的账本数据有不可篡改的特点，这样我们

就可以很方便、很准确地追踪每一笔资金的来龙去脉，如图 1-2 所示。

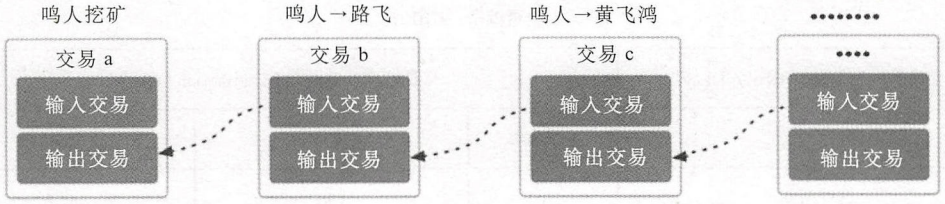


图1-2

UTXO 还具有隐私性，因为一个用户可以拥有很多个比特币地址来充当“收款人地址”，因此很难追查一个用户究竟有多少比特币。UTXO 中的收款地址其实是比特币用户的公钥哈希值，用对应的私钥才能“解锁”。因此，只有拥有这个收款地址的用户才能用自己的私钥来解锁对应地址的钱包，然后将在这个地址中记录的“输出交易”转到另一个用户的收款地址，在这个过程中涉及复杂的数字证书与加解密问题，后面会深入讲解。中本聪考虑到比特币系统在几十年甚至数百年以后可能会产生更多的目前无法预见的交易类型，而不是简单的转账交易，因此在交易模块中引入了脚本系统以增强系统的适用性，我们可以将这视为数字货币系统中最早的“智能合约”，其之后在以太坊（Ethereum）及超级账本中被进一步发扬光大。

3. 小结

在比特币平台中，当前的每时每刻都存在造币（挖矿）与交易（Transaction）两种行为，比特币平台中的交易与银行转账及支付宝转账在本质上没有什么不同，都是用户 A 转账给用户 B，但比特币平台有以下几个明显特征。

- 交易使用的货币是比特币。
- 交易数据采用了高强度的数字加密技术，杜绝伪造与欺诈问题。
- 比特币网络是一个 P2P 对等网络，交易过程完全去中心化，不存在第三方的中介机构。
- 交易的手续费由确认该交易合法的矿工获取，人人都有机会参与赚钱。
- 交易的账本数据分布在每个矿工（节点）的矿机上，加入系统的每个节点都可以拉取并在本地保存一份完整的交易记录数据。
- 比特币系统具有很高的容错性，即使大部分节点崩溃，整个比特币平台仍然可以正常运行。

因此，比特币平台是一个完全建立在加密技术、P2P 网络等核心技术上的去中心化的、自动化的数字货币平台。从某种意义上来讲，比特币也可以被看作“软件定义一切（Software Define Everything）”的理念在传统货币金融领域里的一次成功应用。

1.1.2　从比特币到以太坊

比特币开辟了 IT 领域的一个全新天地，吸引了大量的资金与人才，这里不得不提的是出生于 1994 年的俄罗斯天才少年——维塔利克·布特林（Vitalik Buterin）。维塔利克·布特林在高中时开始为《比特币周报》（Bitcoin Weekly）撰写文章，探讨比特币的相关技术并以此赚取比特币稿酬，在 17 岁时创立《比特币杂志》并亲自撰写文章，这奠定了他日后成为比特币意见领袖的地位。2013 年，19 岁的布特林只上了 8 个月的大学课程，便毅然选择了与前辈比尔·盖茨及乔布斯一样的路——退学并投身到轰轰烈烈的比特币革命浪潮中。为了探索比特币技术在加密货币以外的应用，布特林加入了比特币 2.0 时代的转型工作中，与美国、西班牙、意大利等国的比特币开发者社群展开了深入交流。

在比特币 2.0 的研究推进过程中，布特林发现比特币平台存在先天性的设计缺陷，虽然有了脚本系统可以扩展新的交易类型，但这种脚本系统无法支持数字货币以外的其他应用，因此他设想了一个新的通用的区块链平台，在这个平台上所有开发者都可以构建和发布自己的区块链应用，而不仅仅局限在数字货币这一领域，于是在《以太坊白皮书：下一代智能连接与去中心化应用平台》一书中首次提到他的以太坊智能合约平台的最初构想。然而，布特林在满心欢喜地将这一大胆又极具创意的想法告知比特币开发社区，希望能融入现有的比特币区块链平台时，却吃了一个闭门羹，被比特币开发社区无情拒绝。作为同行，笔者很能理解这一结果，在众人眼里：

> 布特林连专科学历都算不上，还很年轻，没写过靠谱的代码，就靠一路“忽悠”，动不动就发表一些让不明真相的吃瓜群众无比激动的毫无技术含量（代码）的文章，就以为自己是行业大牛，异想开天地提出这么不靠谱的想法，还想让开发比特币系统的这些元老工程师（架构师）们被他指挥，真是脑袋被门夹了！

在被比特币开发社区拒绝后，布特林没有选择跳楼或者就此放弃梦想，而是召集了近 20 位伙伴，并投入自己的全部家当——10 万美元奖学金，成立了非营利组织——以太坊基金会。布特林坚持以太坊应该属于所有人，不能被单一企业占有，因此在开发过程中不接受创投投资。2014 年 7 月，以太坊基金会正式启动以太币的众筹募资活动，当时每枚比特币可兑换 2000 枚以太币，造成大轰动，12 小时内热销 700 多万枚以太币，为期 42 天的众

筹让以太坊团队募集到约 31000 枚比特币（市值约 1840 万美元），之后，以太坊的开发在 2015 年年中大功告成。2016 年，22 岁的布特林被《财星》杂志评选为 2016 年 40 岁以下的 40 位杰出人物之一。2018 年年初，一枚以太币的价格首次突破了 1000 美元大关，以太坊市值 1000 亿美元。

布特林首次在以太坊中提出智能合约的概念，这进一步推动了比特币 2.0 时代的发展。2014 年，布特林超越 80 后大神级前辈——Facebook 的扎克伯格，获得 2014 年的世界科技奖！可以说，以太坊开启了区块链技术的新时代，让隐藏在比特币背后不为人知的区块链技术大放异彩，而当区块链这个新技术的独特价值被人们深刻发现和认知以后，一种新的中间件平台——区块链平台应运而生。区块链 2.0 时代的典型代表以太坊和区块链 3.0 时代的典型代表超级账本项目（Hyperledger），分别代表了区块链的两个重要的发展方向：面向公众的公有链和面向企业的联盟链。

1.1.3 山寨币蜂拥而至

比特币的价格一路狂飙，让不少早期跟风的玩家（投资者）大赚一笔，而传说中的比特币之父——中本聪仅凭借一段代码就轻松成为身价千万的 IT 新贵，更催生了无数想一夜暴富的编程高手，于是模仿比特币的各种山寨币如“雨后春笋般破土而出”。一部分山寨币，在严格意义上说是比特币的竞争者，而非俗称的“山寨货币”，因为它们多少都对比特币做了一些改进，但后来国内出现了很多真正意义上的山寨货币，这些山寨货币直接修改比特币源码的一些参数，就摇身一变成为新的数字货币。到 2016 年上半年，全球共诞生了 600 多种数字货币，市值总和接近 100 亿美元，除了“大师兄”比特币、“二师兄”以太币（ETH），还有“三师弟”瑞波币（XRP）与“四师弟”莱特币（LTC）紧随其后。2017 年是数字货币爆发的一年，截至 2017 年年底，全球数字货币的总市值一度突破 6000 亿美元，年度最大涨幅达到 3497.98%，数字货币种类增加至 1334 种，无论是从总市值规模还是从币种数量来看，都实现了爆发式增长。

1. 山寨币是如何进行“山寨”的

山寨币首先“山寨”的是比特币的“挖矿”算法。之前讲到，比特币的“挖矿”是一种拼“蛮力”的工作量证明算法，在游戏刚开始的时候，比特币非常好“挖”，矿工们都用计算机挖矿，各自的速度也差不多，因此一切都很正常且很公平，只需下载比特币相关的客户端软件就可以自动挖矿。随着比特币价格的一路疯长，参与挖矿的人越来越多，因

此使用普通计算机挖矿得到比特币的概率越来越小，有人看到其中的商机，便开始研究能更快地挖矿的专用设备，于是市场上出现了基于 GPU 显卡的挖矿设备，再后来出现了由 ASIC 定制芯片直接执行 HA256 算法的新一代专业矿机——SHA256 矿机，运算速度远超显卡挖矿机。2012 年，美国某挖矿组织宣称将开发出更强力的“挖矿机”，这引发了该国比特币爱好者的不满，因为在比特币的网络规则中，一旦某人掌握了全网 51%的算力，这个人就有了更改区块链记录的“超级权限”，可以否定其他矿工挖矿的账本记录，剽窃其他矿工的劳动成果，这样一来就干扰了比特币世界的公平性。为维护比特币世界的算力平衡，号称我国比特币“四大天王”之一的南瓜张设计了一款名为“阿瓦隆”的超强悍的挖矿机。阿瓦隆也被称为“挖矿神器”，第一批阿瓦隆有 300 台，买到“阿瓦隆”的矿主都大发横财，一台阿瓦隆的价格也曾被炒到 40 万元人民币。在比特币专业矿机出现后，很快就出现了拥有大量专业矿机的“矿场”，这些矿场拥有巨大的挖矿能力，于是比特币的挖矿市场开始被少数公司制玩家把控，个人玩家难以分得一杯羹，而这一切都因为比特币挖矿采用了很简单的 SHA256 算法，为此，比特币的一些研究者开始考虑更为复杂的其他挖矿算法。

2. Tenebrix 币、Fairbrix 币及莱特币

创建于 2011 年的 Tenebrix 币，就是第一个尝试使用 Scrypt 加密算法挖矿的“山寨版”比特币，与 SHA256 算法仅仅消耗 CPU 的功效不同，Scrypt 加密算法在执行的过程中会占用更多的内存，因此，ASIC 定制芯片的矿机就失去优势了，从而实现了更为公平的挖矿竞争。但 Tenebrix 的不足在于其开发者自己也挖币，并且在挖到 700 万枚 Tenebrix 后才正式将 Tenebrix 发布，这种机关算尽的做法引发了很多人的不满。这时，程序员 Charles Lee 模仿 Tenebrix 创建了 Fairbrix 币，并表示自己不会瞒着大家先行挖矿，但不幸的是，在 Fairbrix 软件系统中产生了一个致命的 Bug，使得很多原始块没有了造币能力，加之黑客攻击事件，导致 Fairbrix 币一开始就输在起跑线上。在吸取这次教训之后，Charles Lee 在 2011 年创立了莱特币（Litecoin），成为莱特币之父。相对于比特币，莱特币有以下改进。

- 与比特币的 2100 万枚上限相比，莱特币的上限为 8100 万枚。
- 相对于比特币，莱特币可以提供更快的交易确认机制。
- 相对于比特币，莱特币因为采用了 Scrypt 加密算法，所以更容易在普通计算机上挖掘。

莱特币的价格在飙涨 7500%之后，Charles Lee 居然清仓了！2017 年 12 月 21 日，Charles Lee 在 Reddit 上发帖称，其在最近几天出售了持有的全部莱特币并捐赠：

> 莱特币确实给我带来了很多财富，我现在已经财富自由了，不再需要将我在财务上的成功跟莱特币的成功捆绑在一起，别担心，我没有放弃莱特币，我仍然将所有时间花在莱特币上。当莱特币成功时，我仍然会以各种不同的方式获得奖励，而不是直接通过拥有莱特币。我认为这是继续监督莱特币发展的最好方式。

3. 狗币

在紧跟莱特币之后大获成功的另一个数字货币是狗币（Dogecoin，Doge 的名称源于日本的一只非常有名的柴犬 Kabosu）。狗币诞生于 2013 年 12 月 12 日，它所使用的代码来自莱特币（山寨莱特币），比起昂贵的比特币，它的入门门槛更低，而且更搞笑（狗币的图案主题采用了一只搞笑的柴犬头像）。狗币没有走比特币的赚钱策略，而是很好地利用了美国的“小费文化”，加之狗币数量巨大（第一年挖出 1000 亿枚，以后每年 50 亿枚，没有上限）、价格低廉（2018 年 1 月 8 日的价格为 1 毛钱一枚狗币！）、转账快速（只需一分钟）等特性，使得狗币很适合用于网络打赏等小费支付，参与狗币交易的人都把狗币作为一种表达分享和感恩的方式，因此，狗币在上线一周后就成为美国互联网中第二个受欢迎的“小费电子货币”。到了 2017 年，狗币的市值突然达到了 10 亿美元，要知道狗币的源码自发布以来的两年多里，都没怎么更新过！这次连狗币的创始人 Jackson Palmer 都觉得数字货币绝对有泡沫！

4. 比特币现金

挖矿巨头比特币大陆公司旗下的矿池 ViaBTC 推出比特币现金（Bitcoin Cash，BCC），BCC 修改了比特币的代码，支持大区块（将区块大小提升至 8MB）。2017 年 8 月 1 日 20 时 20 分，BCC 开始挖矿，同时基于比特币网络的原链继续交易，这是比特币的新分支还是另外一种山寨币？业内论调不一。由于 BCC 和比特币同属于一个体系，拥有很多相似之处，如同“孪生兄弟”，所以 BCC 继承了比特币的基础设置、钱包等软件，只需非常简单的改动即可支持；而且由于 BCC 的分配制度，BCC 直接继承了比特币的广大用户，而其他新兴货币只能通过自己的不断宣传来获得用户。正因为 BCC 有很多其他竞争币所没有的优势，所以才一跃成为新生货币市场的霸主，其报价在 2017 年的最高点达到比特币价格的 1/4。美国大型数字货币交易所之一的 Coinbase 在 2017 年年底宣布支持 BCC 交易，这代表 BCC 的应用性、安全性和稳定性逐渐被市场认可，比特币钱包供应商 Bitcoin.com 的联合创始人兼首席技术官 Emil Oldenburg 也曾表示比特币的交易费用过高、交易处理时间过长，自己已经卖出所有比特币，转投到了更高效的 BCC 中。

5. 山寨币与矿机之战

在目前所有基于 POW（工作量证明算法）的数字货币中，采用 SHA-256 算法的只有比特币一家独大，而在采用 Scrypt 系算法的数字货币中，莱特币和狗币几乎占据了全部江山。在商业利益的驱使下，能够执行 Scrypt 算法的基于 ASIC 定制芯片的矿机也诞生了，基本上给绝大多数 Scrypt 币下达了“死亡判决书”。后来出现的基于 X11 算法的数字货币，和莱特币的初心一样，都是为了抵抗 ASIC 矿机扩张所带来的不公平性。X11 挖矿算法最先来自暗黑币（DarkCoin），它的效率比较高，适合 CPU 和 GPU（GPU 效率更优）。相较于采用 Scrypt 系算法，采用 X11 算法的显卡温度在降低 30%以上的同时耗电量降低 30%，显卡不需要更大的电力来挖矿，挖矿成本显著减少，因此 X11 算法是目前对显卡最友好的挖矿算法。同时，X11 算法阻止了 ASIC 矿机在短期内出现，允许 CPU/GPU 的用户挖更长的时间，基于以上实用特性，目前大量的新币种都已开始使用 X11 算法。其中较出名的事件是一个自称 LTC X11 小组的组织对莱特币实施了硬分叉（硬分叉是指在某种数字货币网络中运行旧版本的节点无法识别运行了新版本软件的节点的交易区块数据，导致新旧节点各自延续自己认为正确的区块链，从而分裂成了两条链），在新的莱特币 LTC X11 中采用了 X11 算法，以解决矿机挖矿的不公平问题。实际上，在金钱利益的驱动下，能够执行 X11 算法的基于 ASIC 定制芯片的矿机最终也会出现，这只是时间问题。完全不依靠矿机进行挖矿的恐怕就要数 POS（股权证明）算法了，比如未来币（NXT）不用挖矿，大户向散户不断推广，开钱包就能增加币数，这似乎更像传销了。

1.1.4　不得不提的瑞波币

尽管比特币在 2017 年价格暴涨 13 倍，但在全球的数字货币中，同期内比特币的涨幅只能排名第 14 位。从默默无闻到举世瞩目，排名第一的瑞波币只用了不到一个月，在 2017 年就暴涨 360 倍，在 2017 年央视 CCTV-2 财经频道公布的 350 个资金传销组织名单中就包括瑞波币。据福布斯统计，截至北京时间 2018 年 1 月，瑞波币市值接近 1300 亿美元，在数字货币中排名第二，仅次于比特币，同时，瑞波币的联合创始人和最大股东克里斯·拉森（Chris Larsen）身价已经超过了 590 亿美元，暂时超过了马克·扎克伯格。

事实上，瑞波币的概念早在 2004 年就出现了，远早于比特币（2009 年）。当时，一个名叫 Ryan Fugger 的人想开发一套去中心化的货币支付系统，因此在 2005 年创立了 RipplePay.com，向全球用户提供安全的支付服务。到了 2012 年，克里斯·拉森与 Ripple 的共同创办人一起向 Ryan Fugger 提出数字货币的概念，成立了 Ripple Labs 公司的前身

OpenCoin，宣称可以为大型金融机构提供以区块链技术为基础的国际金融交易支付解决方案（Ripple 支付网络），并在 2013 年 3 月发行了虚拟货币——瑞波币。2015 年，OpenCoin 改名为 Ripple Labs，负责管理瑞波币的发行。由于瑞波币背后有商业公司掌控和推动，所以瑞波币的价值更容易被操控，因此瑞波币看起来更像“传销”。

瑞波币与比特币在本质上是不同的，比特币是一种流通货币，而瑞波币仅限于在 Ripple 支付网络中使用，算是一种辅助工具。Ripple Labs 要求每个 Ripple 账户都至少有 20 个瑞波币，每进行一次交易就会销毁十万分之一个瑞波币，这一费用对于正常交易者来说成本几乎可以忽略不计。由于每次交易都将销毁少量瑞波币，所以这意味着瑞波币的数量会逐渐减少，如果 Ripple 协议能够成为全球主流的支付协议，则交易网关对于瑞波币的需求就会更为广泛，就会导致瑞波币（预期）升值。截至 2017 年，全球有 60 个国家承认 Ripple Labs，有 11 个金融网关正在使用 Ripple 支付网络，美国有 13 家银行可以自由兑换瑞波币，欧洲有 850 家银行和财务专家把瑞波币定位为金融货币。2018 年 1 月，Ripple Labs 官方发布公告：全球顶级的 5 家汇款公司已经计划在 2018 年将瑞波币用于支付。

瑞波币就如同两种法定货币之间的桥梁，企业可以使用瑞波币在全球执行所需的资金流动，且不需要支付额外的费用。当买卖双方都持有瑞波币时，对于像支付学费这样的大笔金额，交易瑞波币确实是一种实用的交易方式，根据 Ripple Labs 官网介绍，瑞波币的一大特色就是“快”，支付结算只需要四秒，与比特币超过一小时和以太币超过两分钟的结算时间相比，堪称目前交易最快的加密货币，比传统的银行系统要快很多。与比特币试图挑战国家的铸币权不同，Ripple 挑战的对象是民间组织团体——SWIFT（环球同业银行金融电讯协会）。SWIFT 是上述跨国转账中 150 元“电讯费”的真正收费者，提供安全、可靠、快捷、标准化、自动化的通信业务，大大加快了银行间的结算速度，目前全球大多数国家的大多数银行都已使用 SWIFT 系统，但代价是银行需要为跨行异地及跨国转账交易支付一笔电讯费给 SWIFT 协会，即跨国转账的手续费。2017 年 6 月，Ripple Labs 首席风险官克瑞格基德（Greg Kidd）这样向腾讯财经描述 Ripple 支付协议可以给我国消费者带来的“福利”：

> 中国家长给自己在美国上学的孩子汇款 1 万美元时，除了需要向银行按 0.1% 的比例支付手续费（约 60 元人民币），还需要向银行支付 150 元人民币的“电讯费”，但如果该银行采用了 Ripple 支付协议的话，就不用支付这 150 元人民币了。

Ripple 这一去中心化的支付清算协议的期望是对抗目前全球银行通用的 SWIFT 协议，如果 Ripple 协议成为金融交易的标准协议，则通过这个支付网络可以转账任意一种货币，包括美元、欧元、人民币、日元甚至比特币，支付就会像收发电子邮件一样快捷、便宜，最重要的是没有所谓的跨行异地及跨国支付费用。

1.1.5　数字加密货币的现状与前景

作为新兴事物，各种数字加密货币不断涌现，一路被各类资金追捧，同时，来自数字加密货币领域的投资回报率不断刷新历史榜单，很多人在极短的时间内暴富。在很多人眼里，这是一个具有无限可能的投资新领域，但也有很多人认为数字加密货币是一场彻头彻尾的数字化骗局，不亚于历史上曾经疯狂过的郁金香，甚至认为数字加密货币是一场打着高科技旗帜席卷全球的新型传销。ICO 众筹就是这样一个充满争议的新事物，它的名字改编自证券界的 Initial Public Offering（首次公开发行）一词，就本质上而言，ICO 也是一种“公开发行”，只是把所发行的标的物由证券变成了数字加密货币。数字加密货币早期的发展过程离不开 ICO 众筹。

可查的最早的数字加密货币众筹 ICO 项目发起于 2013 年 7 月，Mastercoin（MSC，万事达币，现更名为 Omni）通过 Meta-protocol 拓展比特币的功能，募集 5000 枚比特币，用于开发新币种 MSC；2013 年 12 月，未来币募集 21 枚比特币（约等价于当时的 6000 美元）；2014 年 7 月，以太坊 ICO 募集超过 30000 枚比特币并创下历史纪录；2016 年 5 月，TheDAO 众筹等值 1.5 亿美元，破 ICO 世界纪录，但一个月即被黑客攻克，在历史上刻下了深深的双重惊叹号。其他失败的 ICO 虽然没有被人提起，但我们需要明白的是数字加密货币 ICO 的风险很高。国内也有不少 ICO，即所谓的“认购币”，但大部分玩的是“跑得快”的庞氏游戏，其中最为知名的 ICO 项目是 NEO，其前身为“小蚁股”，成立于 2014 年。在监管风波到来之前，NEO 不仅是国内最大的 ICO 项目，同时跻身世界第 12 大数字加密货币，在 2017 年 8 月上旬，NEO 价格激增，一度突破每枚 50 美元，相较年初增长了 500 倍！

央行相关人士在研究大量 ICO 白皮书后得出结论：90%的 ICO 项目涉嫌非法集资和主观故意诈骗，真正募集资金用作项目投资的 ICO 其实连 1%都不到。2017 年 9 月 4 日，中国人民银行等 7 个部门联合发布《关于防范代币发行融资风险的公告》，将 ICO 定性为“一种未经批准非法公开融资的行为”，监管部门同时表示，各类代币发行融资活动应当立即停止，已完成代币发行融资的组织和个人应当做出清退等安排。同年 9 月 8 日，NEO 宣布退币，参与二级交易的投资人不在退还之列。

以比特币为代表的民间发行的数字货币正在全球疯狂蔓延，各国央行和政府对此看法不一。2017 年 6 月，我国央行货币金银局官网发布了一条《关于冒用人民银行名义发行或推广数字货币的风险提示》，全文如下。

近期，个别企业冒用我行名义，将相关数字产品冠以“中国人民银行授权发行”，或是谎称央行发行数字货币推广团队，企图欺骗公众，借机牟取暴利。现就有关情况提示如下：

一、我行尚未发行法定数字货币，也未授权任何机构和企业发行法定数字货币，无推广团队。目前市场上所谓“数字货币”均非法定数字货币。

二、某些机构和企业推出的所谓“数字货币”以及所谓推广央行发行数字货币的行为可能涉及传销和诈骗，请广大公众提高风险意识，理性谨慎投资，防范利益受损。

三、我国的法定货币是人民币。人民币由中国人民银行统一印制、发行。以人民币支付我国境内的一切公共和私人的债务，任何单位和个人不得拒收。请广大公众树立正确的货币观念，爱护人民币，共同维护人民币的正常流通秩序。

早在2013年，我国央行就指明，比特币为“网络虚拟商品”，不是法定货币。而我国98%的比特币交易都是通过比特币交易平台进行的，但这些交易平台风险重重。2017年9月13日，中国互联网金融协会发布《关于防范比特币等所谓“虚拟货币”风险的提示》，在这份提示中明确说明各类所谓“币”的交易平台在我国并无合法设立的依据，比特币等所谓“虚拟货币”缺乏明确的价值基础，市场投机气氛浓厚，价格波动剧烈，投资者盲目跟风炒作，易造成资金损失，投资者需强化风险防范意识。2017年9月，我国监管当局要求境内比特币交易所制定无风险清退方案，并在同年9月底前关停。2017年9月14日，比特币中国发布公告称即日起停止新用户注册。2017年9月30日，数字资产交易平台停止所有交易业务，其他虚拟货币交易所也纷纷步其后尘，火币网、OKCoin币行、微比特等平台纷纷宣布将停止交易。

我国在宣布并关停国内比特币交易市场后，接下来就是堵住源头的行动了——关闭比特币矿场。2018年1月，我国开始了清理比特币的第二步计划：下令关闭国内的比特币挖矿业务，要求各地方政府“引导”辖内企业“有序退出”挖矿业务，受这一影响最大的公司是我国知名的比特币挖矿公司比特币大陆公司，它于2013年成立，大量制造和销售挖矿专用的芯片和矿机，受益于内蒙古、新疆等偏僻地区的便宜电力和廉价租金，迅速建成了自己的矿机集群，拥有一些很有影响力的比特币矿场。

韩国也是世界上最大的民间比特币交易市场之一，极大影响着主流数字货币的价格。2018年12月28日，韩国政府称，该国将推行额外措施，以监管比特币交易中的投机行为。这些措施包括禁止开设匿名账户，以及进一步讨论关闭数字货币交易所等。受此负面消息

的影响，全球数字货币的价格遭腰斩，随后，近 20 万韩国民众在总统府官网的请愿书上签名，反对虚拟货币交易禁令。2018 年 1 月 22 日，韩国政府表示将对数字货币交易收税，间接承认了数字货币交易所的合法地位。

2018 年 4 月 5 日，印度央行宣布：其监管的金融机构不得再处理数字加密货币业务，并声明：印度央行反复警示包括比特币在内的虚拟货币用户、持有者和交易者，交易这些虚拟货币存在多种风险。鉴于相关风险，央行监管的实体不得为任何交易或结算虚拟货币的个人和企业实体处理和提供相关服务，此决定立即生效。印度央行还表示，已经提供数字加密货币服务的金融机构要在一定期限内终止服务，具体情况将另行公告。印度商业报纸《经济时报》此后评论称，印度国民将再也不能通过银行和电子钱包购买数字加密货币了。如果说银行等央行监管的实体都不得促进数字加密货币的买卖，那么个人将无法通过银行账户向数字加密货币的交易钱包中转款。

日本一开始极力拥抱数字货币，自 2017 年以来，向超过 11 家金融机构颁发了数字货币运营牌照，同年 4 月，日本对相关法律进行了修订，规定涉及数字货币业务的公司必须获得金融监管机构许可证。同时，日本《支付服务法案》正式生效，比特币作为虚拟货币支付手段的合法性得到承认。但到了 2018 年 4 月，日本央行面向公众发布“风险提示”，指出数字货币缺乏央行背书，不是传统货币，本身存在盗窃、投机等风险，并警告公众不要被“高科技形象”的浮云遮眼，也不要被“各种谣言”蒙蔽。日本一直引领币圈风气之先，官方和民间对数字货币都推崇备至，而这次“警告”显露出其态度的微妙变化。

放眼全球，在大多数国家，比特币既不是合法货币，也不是完全不受控制的，而是处于一种中间状态，同时，各国央行时刻提醒用户交易比特币的风险。有些国家虽然表示反对比特币，但是没有正式出台法律禁止，全球“禁止和明确限制比特币”的国家仅在少数，例如孟加拉国、玻利维亚、厄瓜多尔、吉尔吉斯斯坦等国，而一些限制比特币发展的国家允许个人持有比特币，但取缔比特币交易市场并禁止用比特币购买商品与服务。

由于比特币在世界各地的人气持续增长，一些政府逐渐意识到数字货币的潜力和优势，各国央行一方面大力限制或谨慎对待民间发行的各类数字加密货币，一方面在考虑发行自己的官方数字货币——中央银行数字货币（CBDC）的可能性。

展望未来，除了比特币引发的各国考虑发行本国中央银行数字货币的大势，比特币背后的区块链技术也在爆发式发展。目前，区块链技术正在快速探索的领域包括资本市场、金融服务、支付和汇款、衍生品交易、征信管理、政府治理、分享经济、供应链、审计、股票交易、物联网等众多环节。以去中心化为基石的数字货币成为大势所趋。2018 年 1

月 5 日，Facebook 创始人扎克伯格宣布自己在 2018 年的挑战计划是利用加密技术和数字加密货币改造平台，将去中心化技术应用到 Facebook 的服务中，让权力从中心化的集权系统中交还给人们。

1.2 理解区块链的概念

1.2.1 深入理解 Blockchain

比特币的狂潮引发了人们对其底层运行机制和原理的探索，并且衍生出了 Blockchain 这个新概念，Blockchain 被翻译为“区块链”，由此，区块链开启了独立于比特币的研究探索新方向。随着研究的深入，人们逐步意识到区块链这一新技术的独特价值，越来越多的商业服务和互联网应用开始搭上区块链这趟快车。

2018 年，一场让人始料未及又具有颠覆性意义的技术革命疯狂来袭，主角就是区块链。区块链技术被认为是继蒸汽机、电力、互联网之后的下一代颠覆性的核心技术，如果说蒸汽机释放了人们的生产力，电力解决了人们的基本生活需求，互联网彻底改变了信息传递的方式，那么区块链作为建立信任的机器，将可能彻底改变整个人类社会的价值传递方式。区块链技术的热度，从整个互联网和金融行业对比特币的狂热，就可见一斑。Web 浏览器拓荒者 Marc Andressen 指出：

> 在 20 年后，我们就会像讨论今天的互联网一样，讨论区块链。

Blockchain 顾名思义，就是由一个个 Block（块）串联成的一个链条，在每个 Block 里保存的是一笔笔交易（Transaction）数据，以比特币为例，每隔十分钟左右就会有矿机挖出新币。之前说过，挖出新币也是一种交易，被称为“Coinbase 交易”，它作为这段时间内的第一笔交易，会连同在这段时间内产生的其他交易被“打包”为一个 Block，每个 Block 与前后紧邻的 Block 相互链接，组成一个链表。从数据结构上来看，Blockchain 非常类似于我们熟悉的传统链表的数据结构，如图 1-3 所示。

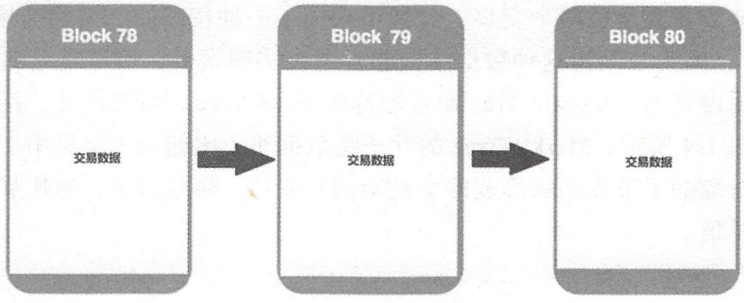


图1-3

但 Blockchain 与普通的链表又有区别，因为链表一般是“水平”视角的，比如从左到右增加长度，Blockchain 则是“上下”视角，从下到上一层层叠加 Block，每增加一个 Block，则 Blockchain 的高度+1。从另一个角度来看，区块链的高度代表区块链数据的“版本”，比如在高度为 2 的区块链与高度为 3 的区块链之间相差一个版本，如果某个节点本地记录的区块链高度为 2，而与之相连的某个节点的区块链高度为 3，则此节点只要从邻居节点那里拉取第 3 个 Block 数据，即可完成账本的同步。

那么，Blockchain 设计的独特之处在哪里呢？

首先，Blockchain 记录的交易具有很强的公信力，具备可信任与防篡改特性。可信任这一点是通过数字证书机制来保证的。以比特币为例，我们知道一个用户可用的比特币余额对应的是 UTXO，这笔钱存放在该用户的某个比特币钱包地址中，并且这个地址只能用对应的私钥打开（解密），因此，只有该用户才能动用这笔钱来发起一个新的交易，在发起的新交易中，该用户的私钥签名信息也一同被登记到交易记录中，在有了数字证书签名后，在随后的传输及持久化保存到 Block 的过程中，任何节点都无法篡改这笔交易的记录，并且任一节点都可以通过该用户公开的公钥来验证这笔交易的合法性，然后予以承认并记录到账本中。此外，Blockchain 里的每个 Block 除了用指针链条来记录前后关系，还用了基于哈希摘要的算法来加强这个链条，具体做法是在每个新生成的 Block 里都记录前一个 Block 内容的哈希值（32 字节）。我们知道，由于哈希算法的特殊性，输入原文的任一字节发生变化，都会导致哈希值不同，因而任一 Block 的内容在被伪造变动后，它的哈希值都会发生变化，后续所有 Block 都必须继续伪造，从而引发连锁效应，这显然是极其困难的。

其次，在比特币的 Blockchain 里还自带“索引”设计。在比特币的平台里，平均每个交易的数据为 250 字节左右，每个 Block 平均包含 2000 个 Transaction，为了能在这 2000

多个交易中快速查找和定位某个交易，比特币采用了一种特殊的二叉树结构 Merkle Tree 来实现“索引”，因为二叉树这种数据结构非常适合实现索引，所以常见的关系型数据库普遍采用二叉树做索引。Merkle Tree 通常被称作 Hash Tree，顾名思义，就是存储哈希值的一棵树。如图 1-4 所示，Merkle Tree 的叶子是数据块（比如一个字符串）的哈希值，而非叶节点是其对应的子节点串联形成的字符串的哈希值，简单来说，非叶节点总是所有子节点内容的哈希值。

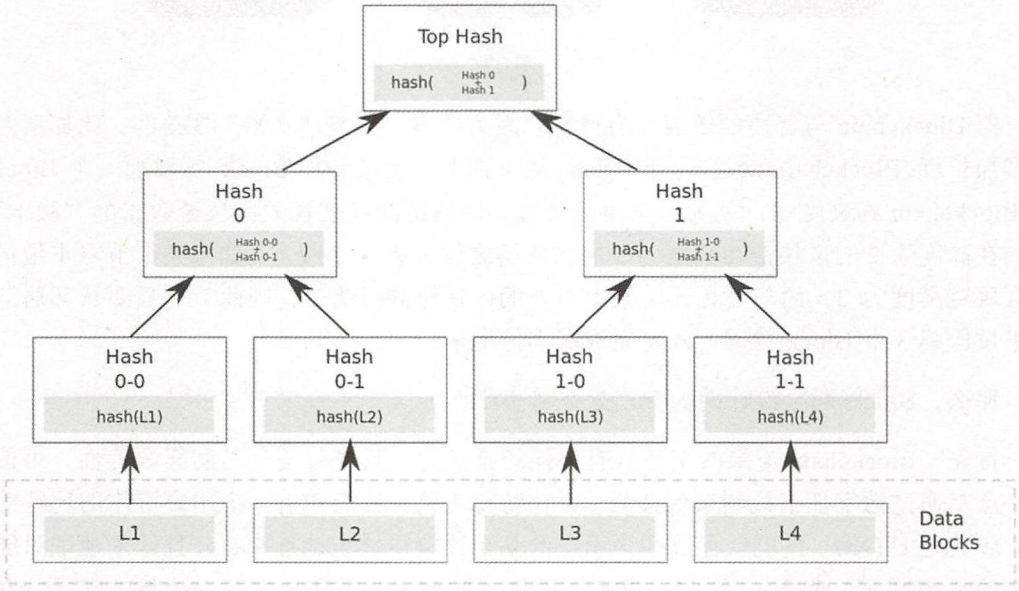


图1-4

我们可以对 Block 中的每个交易计算 32 字节的哈希值，然后把它们当作所有叶子节点，以此构建出一棵完整的 Merkle Tree。在比特币平台上每个交易的数据平均为 250 字节，可以看出，Merkle Tree 相当于把一个 Block 压缩到了 1/8 大小，这样一来，客户端程序如果只想验证某个交易是否完成或存在，则无须下载完整的 Block 数据，只需获取这个 Block 里的 Merkle Tree，对目标交易计算哈希值，然后将哈希结果与 Merkle Tree 中各个中间节点的哈希值依次进行对比，就能够很轻松地确定这个交易是否存在，以及位于哪个叶子节点上，这对于不参与比特币挖矿的轻量级客户端来说尤为关键。图 1-5 和图 1-6 给出了包括 Merkle Tree 在内的 Blockchain 的完整示意图。

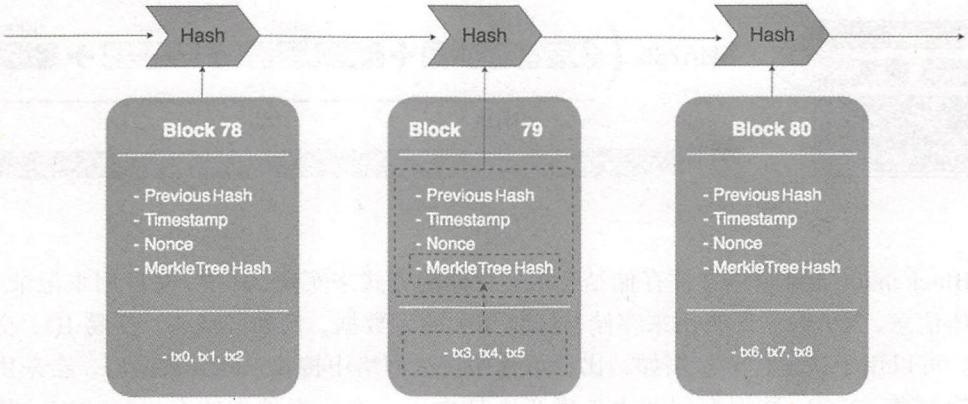


图1-5

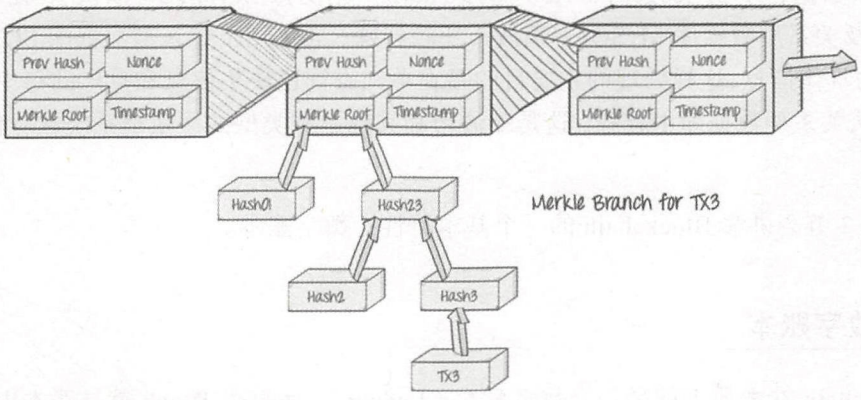


图1-6

从图 1-5 和图 1-6 可以看出，在比特币的每个 Block 里还记录了一个叫作 Nonce 的字段，是 4 个字节的整数。前面讲到，比特币挖矿就是寻找这个 Nonce 随机数的过程。Nonce 是 Number once 的缩写，在密码学中是一个只被使用了一次的随机数值，在加密技术中初始向量和加密散列函数都发挥着重要的作用。挖矿就是要计算出新 Block 的哈希值，计算这个哈希值的输入参数有：比特币版本号、前一个 Block 的哈希值、Merkle Tree 的根、区块时间戳及 Nonce 等数值，基本上可变的参数只有 Nonce，因此比特币矿机需要不断修改 Nonce 的值，才可能计算出符合条件的目标哈希值，这个目标哈希值要小于某个指定的阈值——nBits，因此 nBits 字段也被称为挖矿难度系数，它也存在于 Block 的 Header 字段中，图 1-7 很形象地给出了挖矿过程涉及的上述参数。

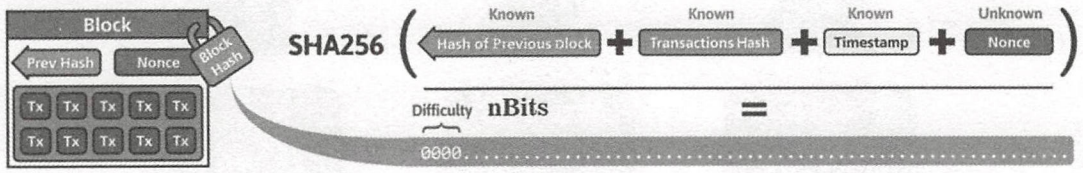


图1-7

Blockchain 通常采用文件存储结合 KV 存储的方式来实现，其中，文件用来记录 Block 的具体信息，KV 数据库则用来存储与区块相关的元数据，例如区块头、交易 ID、交易状态等，可以用于快速检索。例如，比特币程序在从网络中接收 Block 数据后，会采用网络格式直接将 Block 数据存储在本地磁盘文件中（一个块文件大约有 128MB），并采用 LevelDB 作为区块元数据的存储模块，LevelDB 存储了 Block 索引与元数据相关的内容，以加速检索数据；Hyperledger Fabric（简称 Fabric）则支持 LevelDB（默认）、独立部署的 CouchDB 及关系型数据库 MySQL；瑞波币比较特殊，它采用了关系型数据库 SQLite 来存储 Block 的具体信息，这样处理的主要目的是在单纯查询区块头信息和具体的每笔交易时，可以直接从关系型数据库中查找，这是瑞波币和其他三种类型的区块链系统在存储方面的最大不同。

在 1.2.2 节会讲解 Blockchain 的一个基本特性：数字账本。

1.2.2 数字账本

Blockchain 在本质上就是一个数字账本（Ledger），而每个 Block 就是账本中的一页，任一 Block 被修改或者“丢失”都会导致整个链条无效，这与传统的账本在功能和作用上是完全一致的。此外，在数字化账本以后，对账问题不再是一个令人头疼的问题了，以编程方式随时可以精确对账，也更容易快速分析和追踪账务的往来信息，财务造假及洗钱将变得很难，收税也会变得更加方便。

既然比特币宣称是一个去中心化的数字货币系统，那么其中最为关键的基础数据——“账本”肯定是要分布式存储的，并且账本所采用的分布式技术一定与传统的数据分布式技术有所不同，因为它是账本数据，需要高度完整、绝对安全可靠并且高度一致。所以，比特币的账本采用了“一致性的多副本机制”，即比特币网络中的每个矿工节点（Miner Peer）都需要在本地独立保存一份完整的账本数据（Blockchain），如图 1-8 所示。

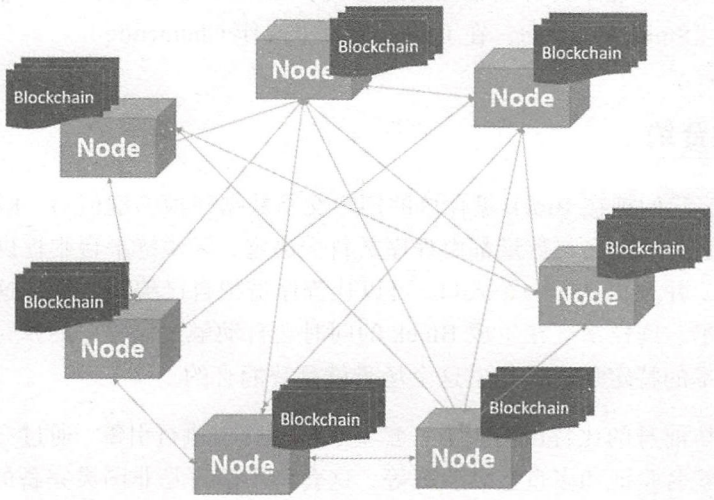


图1-8

既然是分布式的账本，那么问题来了：比特币网络中的某个节点在宕机一段时间后，重新加入比特币网络时，它的账本数据就缺失了，如何重新同步？与此相关的另外一个问题：加入比特币网络的新节点如何获取之前的所有账本记录？

答案很简单，因为 Block 一旦生成就不可变，并且组成账本的基本单元是 Block，因此客户端可以告诉服务器，跳过某个 Block 编号之前的数据，从而实现简单的“断点续传”同步功能。此外，每隔 10 分钟才生成一个新的 Block，因此不用太频繁地同步，也能很快保持全网账本的数据一致性。

区块链技术的这种“分布式账本”功能，不但可以监控资金及数字货币的转移，同样可以监控任何有价资产的交易，例如股票、债券、期货、不动产、珠宝等，目前这些交易大部分都需要由中心化的银行或金融机构来负责记录，也导致了额外的时间和成本。一些研究机构的分析师预测：区块链技术能够在五年之内为全球交易的清算和结算节省超过 160 亿美元，并且分布式账本模式为金融市场带来的影响远不止简化流程这么简单。甚至有人认为：分布式账本模式将会颠覆华尔街的金融业。此外，分布式账本技术也将给会计、审计行业带来颠覆性的改变，已经使用了几百年的复式记账法慢慢地不再适应数字化社会的需要，新的记账机制需要与分布式账本技术有机结合，一定会带来更高的效率和更低的成本，以及更多的被迫改行的会计。

既然要做通用化的数字账本，那么 Blockchain 就不能只存放比特币的交易记录了，是

否有一种机制可以让程序员自主编程决定在 Blockchain 里存放什么交易记录？这个机制就是智能合约（Smart Contract，在 Fabric 项目里叫作 Chaincode）。

1.2.3 智能合约

设想一下，我们把在 Block 里存放的用户交易数据建模为键值对（Key/Value）的一个集合，这些键值对的定义与数量都由程序员自主决定，区块链平台则提供了针对这些数据的 CRUD 接口，并且提供了一个入口，可以让程序员把自己编写的操作这些键值对的一段程序嵌入平台里，这样系统在生成 Block 的同时会自动触发该程序的执行，从而完成将用户数据写入账本的特定逻辑操作，这个场景就是智能合约。

作为以太坊前身的比特币内置了一套基于栈的脚本执行引擎，通过一段脚本对交易进行验证，比如签名验证和多重签名验证等，这套脚本语言是非图灵完备的（图灵完备通常指具有无限存储能力的通用物理机器或编程语言），足够简单却也足以应对货币转账的各种需求。而以太坊的目标是将区块链发展成可以承载更多业务而非只能承载单纯的数字货币的通用型平台，因此它首先提出并实现了区块链 2.0 的核心特性——智能合约。以太坊的智能合约是首个具备图灵完备能力的智能合约，简单来说，能计算一切可计算的问题，比如可以执行判断、无限循环等指令，这样的编程语言就叫作图灵完备的语言。有了智能合约后，用户便可以在以太坊的平台上创建自己的合约，而合约的内容可以包含货币转账在内的任意逻辑。也正是因为区块链 2.0 中智能合约的出现，让区块链从一项小众技术演变成一个热点平台。

智能合约这个概念在很早之前就有了，但在区块链出现后才开始进行有实际价值的应用。下面以常规的银行账号与转账流程这个业务场景为例，来说明与 Fabric 智能合约相关的一些概念与机制。假设有两个账号 A 与 B，A 账号的初始值为 100 元，B 账号的初始值为 50 元，那么可以通过以下 KV 数据操作来记录这两个账号的账本信息：

```
Set        A=100
Set        B=50
```

此时，在 Blockchain 里就产生了一个交易，交易记录的内容为：A=100，B=50。接下来编写一个智能合约的函数来完成转账操作，这个函数很简单，方法签名如下：

```
void transferTo(from,to , amount)
```

用伪代码实现的转账逻辑也很简单（省略了校验）：

```
fromAccount=getValue(from)
toAccount=getValue(to)
setValue(from, fromAccount-amount)
setValue(to, toAccount+amount)
```

在用户触发 transferTo(A,B,50)这个智能合约函数的调用之后，A 与 B 的账号余额就发生了改变，并且被持久记录到 Blockchain 里，扩散到全网。在 Fabric 智能合约里，这些 KV 键值对的当前最新值组成了 World State（世界态），所有 KV 键值对都被存放在一个 KV 数据库里，目前默认使用的是 Level DB。有了 World State 这个扩展，智能合约的编写就变得更加简单，因为我们不需要遍历所有交易记录来计算某个 Key 的最新值。

要实现 Fabric 上的智能合约，则需要经过以下步骤。

（1）编写智能合约代码（用 Go 或者 Java 语言）。

（2）部署智能合约到 Fabric 网络节点中。

（3）初始化智能合约（相当于 Set A=100 及 Set B=50 的操作）。

（4）调用智能合约，例如 transferTo(from,to,amount)。

以太坊里的智能合约采用了一种类似 JavaScript 的编程语言 Solidity，运行在以太坊虚拟机（EVM）中。EVM 提供了堆栈、内存、存储器等虚拟硬件，以及一套专用的指令集，所有代码都在沙盒中运行；还提供了合约间相互调用的能力，甚至可以在运行时动态加载其他合约的代码来执行，这种能力使得以太坊的合约具有非常高的灵活性，但也可能会使合约的功能具有不确定性。

而 Fabric 上的智能合约也是运行在一个类似的沙箱环境即 Docker 容器中的，这种设计的重要出发点是系统的安全性，通过进程隔离，避免用户的智能合约代码系统入侵平台，引发重大安全漏洞。

在去中心化的分布式网络环境中，每个节点都有权利和机会产生一个新的 Block 并要求记入公共账本中，在这种情况下，如何让大家相信 Block 的发起人非常可靠呢？如何确保账本数据不会被“一小撮别有用心的人”肆意编造呢？这就涉及区块链中另外一个复杂的话题——共识机制（Consensus Mechanism）。

1.2.4　共识机制

什么是“共识机制”？共识机制或者共识算法处理的是大家针对某个 Proposal（提案）

达成一致意见（同意提案或者否定提案）的过程。Proposal 在分布式系统中是一个十分宽泛的话题，例如多个事件发生的先后顺序、多副本情况下某个键对应的最终值、集群中谁是 Leader，等等，这些需要集群中多个节点“达成共识”的话题都可以被看作一个 Proposal。区块链上的共识机制主要解决由谁来构造区块，以及如何维护区块链统一的问题。

如果分布式系统中的各个节点都能保证以十分强大的性能（瞬间响应、高吞吐）无故障地运行，则实现共识过程的逻辑并不复杂，只需简单地通过多波过程投票即可。但在现实中这样“完美”的系统并不存在，例如响应请求往往存在时延、高峰期可能临时假死且不响应请求、网络会发生中断、节点会发生故障，甚至存在恶意节点故意破坏系统，等等，因此在实际的分布式系统中要实现高效的共识机制是非常复杂的数学及编程问题。

我们可以将分布式集群中的节点错误问题归为两类：在一般情况下，把节点故障（不响应）的情况称为非拜占庭错误，把节点恶意响应的情况称为拜占庭错误（Byzantine Failures）。拜占庭将军问题的最初描述是：N 个将军被分隔在不同的地方，希望通过某种协议达成某个命令的一致性（比如一起进攻或者一起后退），但其中一些叛徒会通过发送错误的消息阻挠忠诚的将军达成命令上的一致性。莱斯利·兰伯特证明在将军总数 n 超过叛徒总数 m 的 3 倍时，即满足 $n \geq 3m+1$ 时，可以达成命令上的一致性，否则无法达成。简单解释如下：

> 假设只有 3 个将军：A、B、C，三人中有一个是叛徒。当 A 发出进攻命令时，B 如果是叛徒，他便可能告诉 C，他收到的是“撤退”的命令。这时 C 收到一个“进攻”的命令和一个“撤退”的命令，于是被信息所迷惑，无所适从。因此在只有三个将军的系统中，只要有一个是叛徒，拜占庭将军问题便不可解。

在一个由互不信任的各个组织（个人）所构成的分布式网络中，要获得最大的利益，就必须一起努力才能完成，如何达成一致的共识就变成了一个难题，因为根本不知道谁是可信任的，特别是在一个“商场如战场”的金钱帝国里。莱斯利·兰伯特提出了拜占庭将军问题，但真正第一次巧妙解决这个问题的人是中本聪，中本聪的解题思路如下。

（1）首先，如果运行系统中的每个“将军”都同时发出自己的提议，便会产生大量的通信，大大增加了达成一致性的困难，因此，中本聪巧妙地在比特币系统中加入了节点发送提议的成本，即通过挖矿方式的工作量证明机制（Proof of Work，PoW），使得在一段时间内只有一个节点可以发出提议。PoW 机制同时巧妙地解决了集群中陌生节点之间的信任问题——一个努力工作的人要比游手好闲的人可靠得多，现实中的毕业证、五一劳动奖章等都属于工作量证明，它证明我们过去付出了多少努力。在拜占庭系统里加入工作量证明，

其实就是简单粗暴地引入了一个条件：大家都别忙着发出提议，都来做个难题，看谁付出最大的努力去解决了这个难题，谁就有资格第一个发出提议并被大家认可。如果不同的将军先后解出了这个难题，那怎么办呢？只有最早解出难题的将军发出的提议才是有效的，因此，我们看到在比特币的 Block 里有一个时间戳字段。

（2）其次，用非对称加密技术将一个不可信的分布式网络变成了一个可信的网络。比特币网络中的每个节点都有一对公私钥证书，任何节点发出的消息都可以用自己的私钥签名，其他节点无法伪造这个签名，并可以用消息发送方的公钥验证这段消息的真伪并且确定消息发送方的真实身份。由此，一个不可信的分布式网络变成了一个可信的网络，所有参与者都可以在某件事上达成一致了。在比特币这个分布式网络里：

- 每个将军（节点）都有一份与其他将军实时同步的消息账本；
- 在账本里有每个将军的签名，都是可以验证身份的。如果有消息不一致，就可以知道是哪些将军的消息不一致；
- 尽管有消息不一致的现象发生，但只要超过一半的将军同意进攻（生成区块），则少数服从多数，达成共识。在比特币网络里，想要掌握控制权和改写交易规则，则必须掌握全网 51%的计算力，面对比特币现在的网络规模，这已经很难做到了。

中本聪在比特币中其实是“绕过”了拜占庭算法的。原始的拜占庭算法效率很低，算法复杂度为指数级。1999 年，Miguel Castro 和 Barbara Liskov 提出了改进版的拜占庭算法 PBFT（Practical Byzantine Fault Tolerance），将算法复杂度由原来的指数级降低到多项式级，使得拜占庭容错算法在实际的系统应用中变得可行，其中 Barbara Liskov 就是提出著名的里氏替换原则（LSP）的人，为 2008 年图灵奖得主。在 PBFT 的论文中，作者使用这个算法实现了一个很厉害的具备拜占庭容错特性的网络文件系统（NFS），通过性能测试证明了该系统仅比无副本复制的标准 NFS 慢 3%。Fabric 在 1.0 版本之前曾实现过基于 PBFT 的共识算法，但在发布 1.0 版本时取消了这一算法，可能出于对算法复杂度与成熟度的考虑。

除了 PoW、PBFT 这两种经典的共识机制，还有以下几种常见的数字货币共识机制。

- 权益证明（PoS）。账本理应由持有最大经济利益的人去维护，是完全的资本主义形式——拥有的钱越多，拥有的权力就越大。以太坊下一代的共识机制 Casper 也属于 PoS 的一种，要求验证人将大部分保证金对共识结果进行下注，验证人必须猜测其他人会赌哪个块儿胜出，同时下注这个块儿，如果赌对了，就可以拿回保证金外加交易费用，也许还会得到一些新发的货币；如果下注没有迅速达成一致，

就只能拿回部分保证金。

- 股份授权证明（DPoS）。引入了“受托人”的角色，所有持币者先选出受托人负责签署区块：选举过程类似于由股东会选举出董事会，代替股东会做出日常运营决策。在授权董事会后，决策会更有效率。
- 瑞波共识机制（Ripple Consensus）。使一组中心化的特殊节点列表达成共识，初始特殊节点列表就像一个俱乐部，要接纳一个新成员，则必须有 51%的成员投票通过，由于该俱乐部从“中心化”开始，所以一直是中心化的系统。
- 基于传统的分布式一致性技术及数据验证机制。在成熟的分布式一致性算法（Paxos、Raft）的基础上实现秒级共识验证，但去中心化程度不如比特币，更适合多方参与的多中心商业模式。

在理解了区块链的基本概念以后，接下来快速体验本书的主角——Fabric 的功能。Fabric 是 IBM 开源并主导的区块链基础设施，或者说是区块链中间件，也是基于区块链 3.0 技术的企业联盟链的代表，采用了模块化的设计思路，因此其成员服务、共识算法、区块链存储等关键模块都能得到灵活扩展；此外，Fabric 还使用了领先的基于 Docker 容器技术的智能合约。综合上述特性，Fabric 可以作为领先的、企业级的、通用的开源区块链平台。

1.3 快速体验 Fabric

1.3.1 Fabric 的概念与术语

区块链平台首先是一个 P2P 结构的区块链网络（Blockchain Network），Fabric 的区块链网络结构如图 1-9 所示。

我们看到，Fabric 的区块链网络在总体上分为两类节点：Fabric Orderer 节点（以下简称 Orderer 节点）及 Fabric Peer 节点（以下简称 Peer 节点），其中 Orderer 节点主要负责解决区块链交易中的共识机制问题，Peer 节点则主要负责保存区块链账本数据。我们在客户端（Submitting Client，为 Fabric 命令行工具或者通过 Fabric SDK 编写的客户端程序）发起一个区块链交易后，就可以将此交易提交到任意一个或多个 Peer 节点上，这些 Peer 节点在完成必要的交易验证逻辑（如交易者的身份核实及数字签名的真伪验证）后，再将交易数据发送到一个或多个 Orderer 节点，再经过 Orderer 节点特定的共识算法验证筛选后，

合规的交易被 Orderer 节点打包成一个 Block，随后被广播传输到整个 Peer 节点网络中，最终被存储为持久化的账本数据（Ledger）。

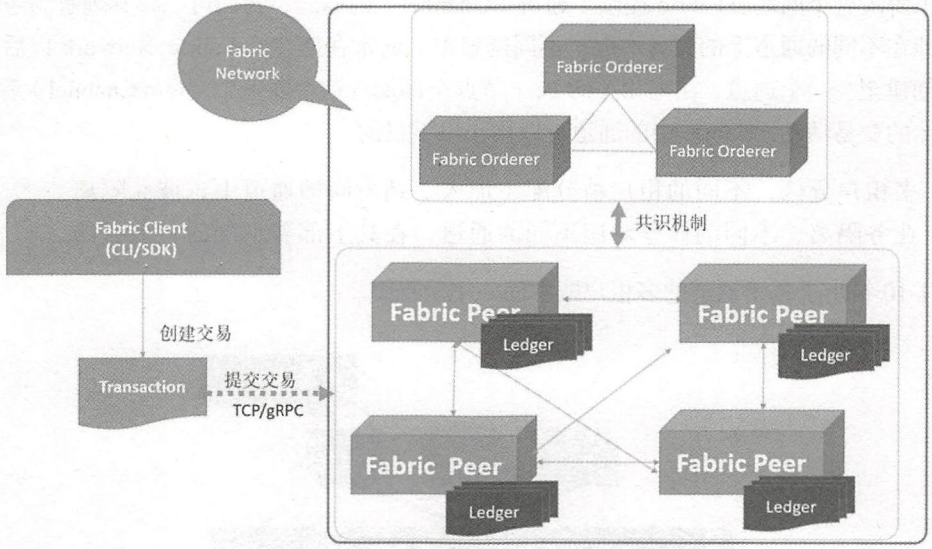


图1-9

与比特币这种任何人都可以加入的完全公开的区块链网络不同，Fabric 网络中的所有 Peer 节点都来自同一个“联盟”，这个联盟通常由一些有着共同商业利益关系的企业组织（Org）组成，这些组织在加入 Fabric 企业联盟链后，就变成联盟中的一个成员（Member），因此在 Fabric 里，成员与组织是等价的两个术语，每个成员都“安插”了一个或多个 Peer 节点接入联盟的 Fabric Network 中，因此，每个联盟中的每个组织都能保存完整的账本数据，并且共同参与到区块链的交易中。那么，Fabric 是如何识别和验证一个 Peer 节点的真实身份及其所属组织的呢？这就要通过数字证书技术来实现了，每个节点（包括 Orderer 节点、Peer 节点及 Fabric Client）都有自己的身份 ID、CA 证书，包括 Peer 身份认证、证书签发、签名验证在内的服务被抽象出来并且被命名为 MSP（Membership Service Provider），MSP 也可被简单理解为“用户身份认证系统”，只不过这里的用户指的是组织。在 Fabric 中，可以是所有组织共用一个 MSP，也可以是每个组织独立用一个 MSP，甚至在更复杂的情况下，可以在一个组织里的多个部门（Department）中分别采用不同的 MSP。不管组织与 MSP 如何映射，我们需要知道的是一个 Peer 节点只能由一个 MSP 拥有，并且无法识别来自 MSP 的其他 Peer 节点，即使它们属于同一个组织。

既然 Fabric 是一个企业联盟链，那么这个链条中的不同 Peer 节点便代表不同的企业组织，

也就是说大家在存在共同的商业利益的同时，又存在商业竞争和商业机密的问题。举例来说，某个业务交易仅限于联盟中的某几个企业之间，这些交易和账本数据应该不能被其他节点所看到。为了解决这个问题，Fabric 提供了通道（Channel）的概念，我们可以将其理解为逻辑分区账本，即在不同的通道上的交易形成了不同的账本。通常在搭建好 Fabric Network 以后，就需要考虑创建至少一个通道，在把相关的 Peer 节点全部加入这个通道中（Join Channel）后，才能发起具体的交易活动。Fabric 中的通道可以用于以下目的。

◎ 多租户分区，不同的租户被分配（加入）到不同的通道中，彼此隔离。
◎ 业务隔离，不同的业务对应不同的通道，在其上部署不同的智能合约。

图 1-10 给出了基于通道的多租户账本的一个示意图。

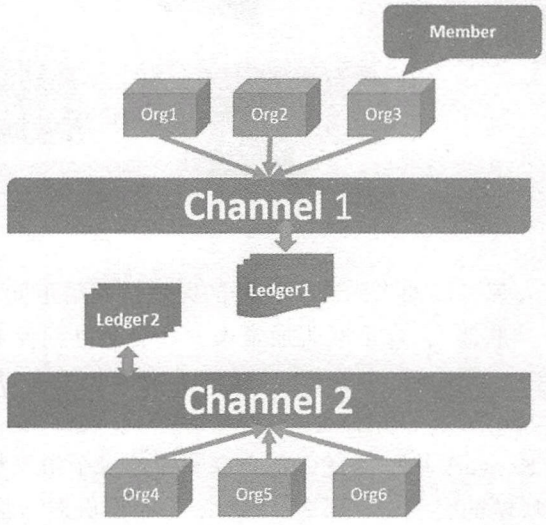


图1-10

如果认真思考任意组织的运作过程，就会发现一个普世规则：任意企业都拥有一个“决策”机构，一些重要的商务活动或交易合同需要经过这些决策机构的审核并签字才能被放行通过。对于牵扯更多企业的区块链交易，是不是也需要类似的交易审核机制呢？于是，我们看到在 Fabric 中出现了一个新的术语——Endorser。Endorser 是“背书”的意思，可以将其理解为“签名盖章”，Fabric 网络中的某些 Peer 节点可以作为背书节点（Endorser Peer），一起来为某个交易请求做“背书”，当然也可以拒绝某些不符合条件的交易。Fabric 允许我们为某些特定类型的交易设置指定的背书策略（Endorsement Policy），比如，规定

所有的大额转账提现交易都必须得到 Org1 与 Org2 的背书（签名盖章）才能进行下去，此时，Org1 与 Org2 中的一些 Peer 节点就承担了 Endorser Peer 的角色，而所有保存交易账本的 Peer 节点就是 Commiter Peer 了。

下面说说另一个问题：Fabric 中的交易（Transaction）。实际上，Fabric 客户端程序是无法直接提交一个交易的，它只能发起一个交易提案（Transaction Proposal），如果存在对应的交易背书策略，这个交易提案就会被转发给背书策略指定的 Endorser Peer 节点去背书，在背书得到一致通过后，交易提案被发送到 Orderer 节点并在这里最终生成交易（Transaction），被打包到区块中，随后被广播到所有 Fabric Peer 节点上并最终持久化保存到这些节点的本地账本（Ledger）中。如图 1-11 所示是 Fabric 中的交易流程。

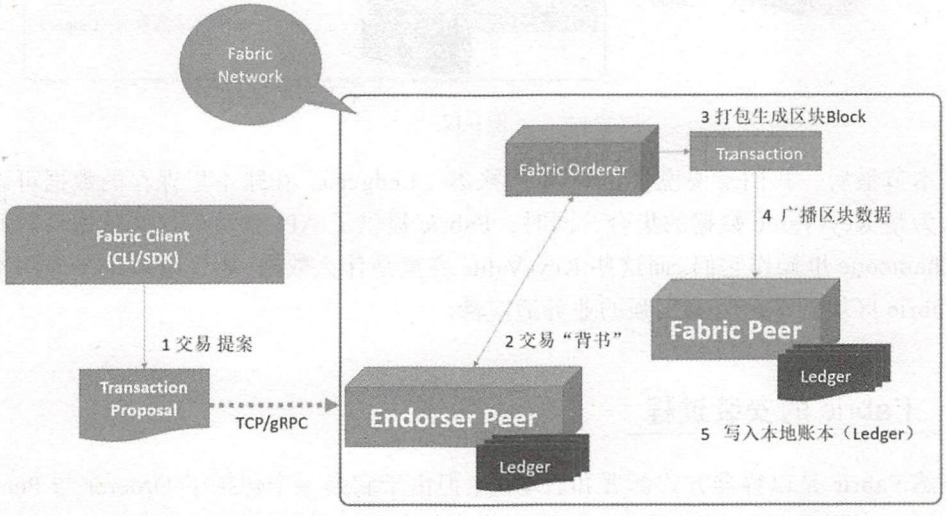


图1-11

发起一个 Transaction Proposal，其实是触发了 Fabric 中的某个智能合约（Chaincode）的调用。Chaincode 是一段运行在 Docker 容器中的符合特定接口的代码，需要经过编译（Compile）、安装（Install）及初始化（Instantiate）这样的流程后才能正常提供服务，此时每个 Chaincode 实例就以一个 Docker 容器的方式存在，通过 gRPC 接口与本机的 Peer 节点进行通信，接收本机 Peer 节点发出的 Transaction Proposal 并执行内部的 Chaincode 代码。增加了智能合约的交易流程如图 1-12 所示。

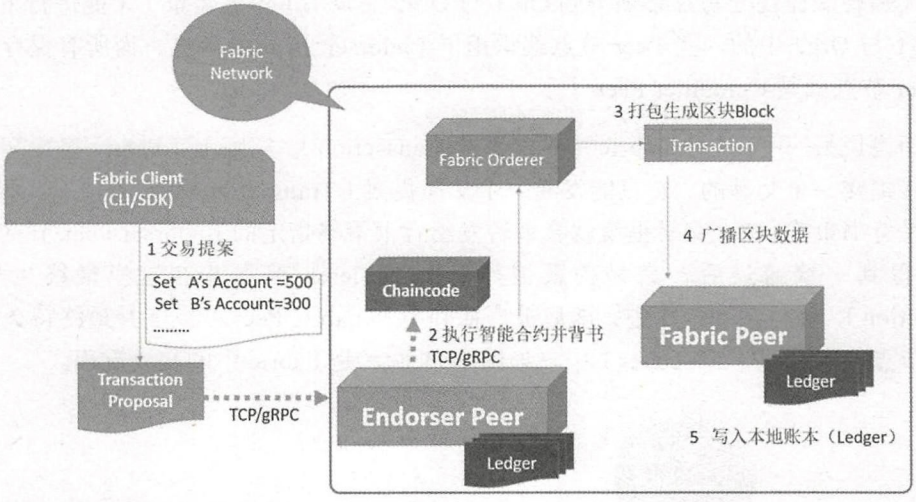


图1-12

在本节最后，我们来说说 Fabric 里的账本（Ledger）。在账本里保存的数据可以被简单地认为是 Key-Value 数据的集合，同时，Fabric 提供了 API 来访问这些数据，我们很容易在 Chaincode 里操作它们。而这些 Key-Value 究竟是什么数据，则完全取决于应用本身，因此 Fabric 区块链平台具有很强的业务适应性。

1.3.2 Fabric 的安装过程

虽然 Fabric 是以容器方式部署和启动的，但由于它是一个包括了 Orderer 与 Peer 两种角色的 P2P 网络集群，同时需要定义这些节点分别属于哪些组织与 MSP，还需要为每个节点生成数字证书来确保安全性，因此 Fabric 的安装部署相对来说还是很复杂的。本节主要对安装过程中的主要步骤和目的做出解释说明，这样，我们就能更容易地分析、定位并解决 Fabric 在各种复杂生产环境下部署时所可能产生的问题。

部署的第一步，是确定 Fabric 网络由几个组织组成，定义每个组织所包含的节点（Orderer 节点或者 Peer 节点）信息，这一步的成果主要是一个名为 configtx.yaml 的模板文件，大概的内容如下：

```
# ---------------------------------------------------------------------------
# "OrdererOrgs" - Definition of organizations managing orderer nodes
# ---------------------------------------------------------------------------
```

```
OrdererOrgs:
  # ---------------------------------------------------------------------------
  # Orderer
  # ---------------------------------------------------------------------------
  - Name: Orderer
    Domain: example.com
    Specs:
      - Hostname: orderer
# ---------------------------------------------------------------------------
# "PeerOrgs" - Definition of organizations managing peer nodes
# ---------------------------------------------------------------------------
PeerOrgs:
  # ---------------------------------------------------------------------------
  # Org1
  # ---------------------------------------------------------------------------
  - Name: Org1
    Domain: org1.example.com
    Template:
      Count: 2
    Users:
      Count: 1
  # ---------------------------------------------------------------------------
  # Org2: See "Org1" for full specification
  # ---------------------------------------------------------------------------
  - Name: Org2
    Domain: org2.example.com
    Template:
      Count: 2
    Users:
      Count: 1
```

上面这个模板文件的第 1 段（OrdererOrgs）定义了管理 Orderer 节点的组织机构，在该组织里只有一个域名为 example.com 的成员，成员的名字为 Orderer（Name: Orderer），包含了一个主机为 orderer 的 Orderer 节点（Hostname: orderer）节点；模板文件的第 2 段（PeerOrgs）定义了管理 Peer 节点的组织机构，在这个示例中分别定义了名为 Org1（域名 org1.example.com）与 Org2 的两个成员，它们分别拥有两个 Peer 节点（Count: 2）。

我们通过 configtx.yaml 确定了整个 Fabric Network 的结构，接下来的工作是生成每个节点的证书。由于这一部分比较麻烦，工作量大，因此 Fabric 除了提供了一个命令行的简单工具 gencrypt，还提供了一套证书管理工具——Fabric CA Server & Client，提供了节点

证书的生成、签名、撤销等日常工作的标准化功能，这套工具也可以独立使用。

下一步就是生成 Fabric 创世块。每个区块链系统中的第 1 个区块都被称为创世块，在 Fabric 的创世块中包括了用于 MSP 身份认证的节点证书信息，这些信息来源于上面的 configtx.yaml 文件，我们可以采用 configtxgen 命令来生成所需的创世块，这个创世块的名称一般为 orderer.block，这是因为 Orderer 节点的启动和运行需要创世块的数据。

接下来，我们就可以配置和启动 Orderer 节点了。Orderer 节点需要保存一些持久化数据，因此对应的 Docker 容器需要挂载相应的 Volume，此外，相关的服务端口也需要映射出来，以供 Peer 节点和 Fabric Client 访问。在 Orderer 节点配置、启动完成后，我们接下来就可以配置、启动各个 Peer 节点了，Peer 节点的配置、启动过程与 Orderer 节点大体相同，这里就不做说明了。

在所有节点都启动完成后，我们接下来需要创建一个通道，然后把相关的各个 Peer 节点加入（peer channel join）这个通道上，以便参与这个通道上的各种交易，接下来就可以部署我们所开发的各种智能合约到这个通道上了。

在具体部署 Fabric 时可采用以下几种部署方式。

- 单机学习版部署。在 Docker 环境中运行，Fabric 网络结构包括一个 Orderer 节点与一个 Peer 节点。
- Docker 多机部署。在多主机集群中部署，可以有多个 Orderer 节点，多个 Peer 节点组成复杂的 Fabric 网络拓扑。
- Docker Swarm 多机部署。与普通的 Docker 多机部署不同，这种方式采用 Docker Swarm 微服务模型建模，部署和配置相对简单。
- Kubernetes 多机部署。类似于 Docker Swarm 集群，但更为强大，建议复杂的大型 Fabric 网络采用这种方式部署。

为了让大家快速上手 Fabric，我们准备了一个单机版学习环境，这个示例可以从 GitHub 网站的 MyCATApache/SuperLedger 项目中下载。

在下载和解压缩示例后，我们得到一个名为 Fabric_Demo 的 VMware 虚拟机，运行环境要求 VMware Fusion 8 以上版本或者 VMware Workstation 12 以上版本。以下是虚拟机的信息。

- CPU 内存：4 核 4GB。
- 硬盘空间：20GB。

- 网络：DHCP。
- 系统：CentOS 7。
- 预制软件：Git、Docker。
- 用户名/密码：root/111111。

我们可以将上述虚拟机导入 VMware 中运行，需要注意的是，为了方便演示，我们关闭了虚拟机的 SELinux 及防火墙，在网络方面出于对兼容性的考虑，使用了 DHCP，如果需要 SSH 登录，则建议修改为静态 IP。

对虚拟机目录的说明如下。

- /hyperledger/scripts：演示脚本所在的目录。
- /hyperledger/scripts/init.sh：Fabric 的初始化脚本。
- /hyperledger/scripts/start.sh：Fabric 的运行脚本。
- /hyperledger/data/config：Fabric 的配置文件目录。
- /hyperledger/data/shared：Fabric 的数据存放目录（在每次运行前都会清空）。
- /user/local/src/go：GOPATH 环境变量的指定目录。
- /user/local/src/go/src/github.com/hyperledger/fabric：Fabric 的源码目录，其中，examples/chaincode/go/chaincode_example02 中的 Chaincode 会被演示代码用到。

在我们的 Fabric_Demo 虚拟机中，Fabric 程序通过 Docker 方式部署和启动。首先运行 init.sh 脚本，会自动下载与 Fabric 相关的 Docker 镜像；接着运行 start.sh 脚本，会完成 Fabric 区块链网络的搭建并部署一些演示用的智能合约；然后就可以在这个环节操作和练习了。下面按照以下步骤进行操作。

首先，进入演示脚本目录：

```
[root@localhost ~]# cd /hyperledger/scripts
```

其次，初始化 Docker：

```
[root@localhost scripts]# ./init.sh
```

在启动之后，脚本会开始拉取 fabric-tools、orderer 与 peer 这三个镜像，注意运行版本是 1.1.0 。

然后，启动 Fabric 网络，执行下面的命令：

```
[root@localhost scripts]# ./start.sh
```


在启动之后，脚本会按顺序执行以下操作。

（1）clear states：清理运行状态（删除 Fabric 的持久化数据）。

（2）clear containers：清理容器。

（3）generate crypto：生成加密证书。

（4）generate genesis：生成创世区块。

（5）start orderer：启动 Orderer 节点。

（6）start peer：启动 Peer 节点。

（7）create channel：创建承载区块链业务的通道 channel1。

（8）join channel：将 Peer 节点添加到 channel1 里。

（9）install chaincode：安装智能合约 example02（转账的智能合约）。

（10）input commands：启动 docker bash。

接下来，我们就可以在上述 Docker 容器中体验 Fabric 区块链的智能合约例子了。

1.3.3 智能合约初体验

首先，我们先说说在上一节 start.sh 脚本中安装的智能合约 example02，这是 Fabric 自带的转账例子的智能合约。在 example02 的例子中，每个账户只有一个名字及对应的余额这两个信息，账户信息可以在 example02 初始化（instantiate）的过程中设置完成。假如要在初始化状态下创建两个账号，分别是 a，对应 100 元；b，对应 200 元，则可以在上一节启动的 Docker 容器中执行以下命令：

```
peer chaincode instantiate -C channel1 -n example02 -v v1 -c '{"Args":["init",
"a","100","b","200"]}'
```

在上述参数中，-C 为通道的名字；-n 为智能合约的名字；-v 为智能合约的版本号；-c 为智能合约的输入参数，为 JSON 格式的字符串。-c 参数的含义为：调用智能合约的 init 方法，以完成 a=100 及 b=200 的账户初始化过程。

接下来就可以调用 example02 的转账接口来实现 a 向 b 转账的功能了，操作起来很简单，转账的命令如下：

```
peer chaincode invoke -C channel1  -n example02 -v v1 -c '{"Args":["invoke
","a","b","10"]}'
```

我们看到，example02 的转账接口调用了 invoke 方法，并且 invoke 方法对应的 3 个参数分别是转出账号、转入账号的名字，以及对应的转账金额。

在执行上述转账命令后，我们就可以执行账号余额查询命令来确认是否转账成功了。首先，查询 a 账号的余额：

```
peer chaincode query -C channel1  -n example02 -v v1 -c '{"Args":["query","a"]}'
```

如果把上述转账与查询的两个命令放到一起，你能否发现里面的“规则”？

```
peer chaincode invoke -C channel1  -n example02 …..
peer chaincode query  -C channel1  -n example02  ….
```

在 Fabric 的命令行客户端中，我们要用“peer chaincode invoke”的方式去调用涉及修改账本的智能合约接口，而对不涉及修改账本的接口则需要用“peer chaincode query”方式去调用，这是为了明确区分交易的类型，因为账本数据一旦被更改，就被永久保存在区块链中，无法再篡改或遮掩。

我们把调用智能合约 example02 的初始化、转账、查询这三个功能的命令参数-c 拿出来对比：

```
"Args":["init","a","100","b","200"]
"Args":["invoke","a","b","10"]
"Args":["query","a"]
```

会得到一个很直观的推论结果：Args 参数是一个 JSON 数组，其中数组的第 1 个元素是目标智能合约的方法名，后继的元素是该方法的参数列表。因此，如果知道一个智能合约的源码，我们就知道如何去调用这个智能合约了。

如下所示是 example02 这个 Chaincode 的入口代码片段：

```
func (t *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
 fmt.Println("ex02 Invoke")
 function, args := stub.GetFunctionAndParameters()
 if function == "invoke" {
     // Make payment of X units from A to B
     return t.invoke(stub, args)
 } else if function == "delete" {
     // Deletes an entity from its state
     return t.delete(stub, args)
```


```
    } else if function == "query" {
        // the old "Query" is now implemtned in invoke
        return t.query(stub, args)
    }
    return shim.Error("Invalid invoke function name. Expecting \"invoke\" \"delete\"
\"query\"")}
```

在上述代码里，GetFunctionAndParameters 方法解析并返回了客户端传递过来的方法签名信息，其中 function 是要调用的方法名，args 是对应的参数列表：

```
function, args := stub.GetFunctionAndParameters()
```

接下来，智能合约的代码会根据用户传过来的方法名（function）转到具体的实现方法中。以实现了转账操作的 invoke 方法为例，其源码主要如下：

```
A = args[0]
B = args[1]
Avalbytes, err := stub.GetState(A)
if err != nil {
    return shim.Error("Failed to get state")
}
if Avalbytes == nil {
    return shim.Error("Entity not found")
}
Aval, _ = strconv.Atoi(string(Avalbytes))

Bvalbytes, err := stub.GetState(B)
if err != nil {
    return shim.Error("Failed to get state")
}
if Bvalbytes == nil {
    return shim.Error("Entity not found")
}
Bval, _ = strconv.Atoi(string(Bvalbytes))
// 执行转账逻辑
X, err = strconv.Atoi(args[2])
if err != nil {
    return shim.Error("Invalid transaction amount, expecting a integer value")
}
Aval = Aval - X
Bval = Bval + X
// 写入账本数据
```


```
    err = stub.PutState(A, []byte(strconv.Itoa(Aval)))
    if err != nil {
        return shim.Error(err.Error())
    }
    err = stub.PutState(B, []byte(strconv.Itoa(Bval)))
    if err != nil {
        return shim.Error(err.Error())
    }
    return shim.Success(nil)
}
```

上述转账代码的逻辑过程如下：

首先，程序会从参数中获取 A、B 两个账号的值；接着，从账本中分别获取这两个账号对应的余额（stub.GetStata）；然后，根据转账金额参数来确定 A、B 账号的新的余额；最后，把新的余额写入账本中。

从这段代码来看，即使转账金额为负数也都能接受，并且在转账过程中没有考虑“空头支票”的问题！考虑到这仅仅是一个示例，所以我们也能接受这个结果。

从 example02 的智能合约源码来看，Fabric 所提供的智能合约是很灵活、很强大的，存储在区块链账本中的数据可以是基于 Key-Value 的任意集合，并且 Key 与 Value 都是原生的字节数组，通过 Fabric 提供的 API，我们可以很方便地操作这些数据。此外，我们可以根据自己的业务需要来定义智能合约的方法签名，不必受限于特定的方法签名。在后面的章节中会继续深入讲解智能合约。

上面给出了 Fabric 的第 1 个智能合约的例子，我们安装和运行了一个模拟银行转账功能的智能合约，在这些操作中，除了智能合约的初始化操作只能执行一次，其他操作如转账操作与余额查询，都可以重复执行若干次。

第 2 章
区块链的生态与原理

2.1 区块链的生态

2.1.1 Hyperledger 社区

由于点对点网络的特性，分布式账本技术是完全共享、透明和去中心化的，所以非常适用于金融、制造、银行、保险、物联网等很多行业。在 IBM 等巨头看来，区块链将会成为至关重要的技术模型，推动众多工业与企业进行革新，因此，在企业级区块链这个技术还没有标准化之前，以开源的高姿态引导业界并且确定此领域实际上的领导地位，是回报颇丰的做法。在 2015 年，IBM 联手 Linux 基金会共同推动了区块链领域的基础项目——Hyperledger（超级账本），超级账本技术指导委员会主席由 IBM 兼任。由于参与者都是各个行业的巨头，并且该项目有望成为未来企业级区块链技术的底层架构基础，所以该项目的进展始终备受关注。

Hyperledger 项目的目标定位是打造企业级区块链框架，做到既高效又可扩展，并且能够为隐私与机密相关的需求提供企业级支持。如果说以比特币为代表的货币区块链技术是区块链 1.0 时代的典型代表，以以太坊为代表的合同区块链技术是区块链 2.0 时代的典型代表，那么实现了完备的权限控制和安全保障的 Hyperledger 项目毫无疑问是区块链 3.0 时代的典型代表。

虽然 Hyperledger 联盟是开源组织，但一个公司需要经过严格的申请程序，以证明其代码的成熟度及承诺资源，才能得到 Hyperledger 联盟的正式认可，从而获得代码的“孵化”资格。在目前的 185 个 Hyperledger 联盟成员中，只有 8 个代码库是获得正式批准的。而在这 8 个代码库中，只有 3 个项目框架被评为“活跃”，它们分别是 IBM 贡献的 Fabric、英特尔的 Sawtooth 项目，以及由日本创业公司 Soramitsu 提交的 Iroha。Iroha 是其他 Hyperledger 基础设施项目（Fabric、Sawtooth 等）的补充，鼓励手机区块链应用程序的开发。

在 Fabric 的代码中有 1/3 被用于专有的 IBM 区块链平台，任何人都可以在其上建立内容，即使他们想要创造的是 IBM 的直接竞争对手。2017 年 7 月，由 IBM 领导的 Fabric 项目正式发布 1.0 版本，这标志着 Hyperledger 联盟的首个开源企业级区块链平台的诞生。目前约有 27 个组织参与了 Fabric 项目并做出贡献。Fabric 1.0 在正式发布之前，已经有几百到上千个基于 Fabric 的 POC 项目了，据悉，此前银联与京东合作的联盟链平台正是构建在 Fabric 0.6 上的。

英特尔的 Sawtooth 涉及一个新的共识算法 PoET，用于证明过往的时间事件（Proof of Elapsed Time），其目标是形成资源消耗最少的绿色环保共识机制。PoET 需要全新的硬件芯片的支持，所以我们可以理解为什么 Intel 会发力这样一个新的区块链项目。Sawtooth Ethereum 项目（Seth）的公布，让以太坊智能合约和 Sawtooth 的结合成为可能——混合“锯齿以太坊”。通过这种整合，现有的以太坊开发者预计能够将自己的工作（以太坊智能合约）过渡到 Sawtooth 平台。2018 年 1 月 30 日，Sawtooth 1.0 正式发布，成为 Hyperledger 联盟旗下第 2 个达到 1.0 正式版的可用平台。在使用 Sawtooth 的早期公司中有通信巨头华为，同时，电商巨头亚马逊也将 Sawtooth 列为它的区块链合作伙伴。

Hyperledger 项目发展至今，目前已在全球拥有 160 多个成员，国内也有不少 Hyperledger 的联盟成员，下面介绍国内的典型成员。

云象区块链成立于 2014 年 10 月，在 2017 年 7 月完成数千万 Pre-A 轮融资，专注于为企业级客户打造商业智能合约应用，也为行业私有链应用提供安全的、部署成本较低的区块链数据库产品。其创始人兼首席执行官黄步认为，Hyperledger 项目与公司的企业级联盟链平台产品有很好的切合度与互赢效应。

北京太一云科技有限公司是新三板第一家区块链上市企业，太一云科技公司创始人兼首席执行官邓迪表示：“太一云科技很荣幸加入 Hyperledger 项目，并应努力扩大 Hyperledger 项目在我国的影响力。太一云科技将积极参与 Hyperledger 社区的活动，推进

区块链生态系统并促进区块链的广泛应用，作为包容型金融和共享经济的基础设施。”

联合移动支付电子商务有限公司是由中国移动与中国银联联合发起并成立的移动支付公司，其创始人兼首席执行官张斌表示：“区块链是未来新经济实现的核心技术，我们坚信 Hyperledger 是区块链中最有前途的项目，能够实现在金融领域的成功。我们很高兴能够成为 Hyperledger 项目的积极参与者，并与其他参与者合作，为金融系统建立新的基础设施。”

2017 年 10 月，百度金融正式加入 Hyperledger 项目，成为该项目的核心董事会成员，百度副总裁张旭阳表示：“区块链技术领域的研究和探索是一项长期战略性投入。加入 Hyperledger 以后，我们期待能与其他成员合作，推动区块链技术解决方案的演进及全球区块链技术开源标准的制定。同时，我们希望利用百度强大的技术场景优势，赋能百度金融消费生态，在继续加速区块链技术应用落地的同时，为更多的合作伙伴输出区块链技术解决方案，实现合作共赢。”

2018 年年初，腾讯云正式加入 Hyperledger 项目，表示将深度参与国际区块链生态建设，参与并推动区块链技术及相关标准的制定。此前，腾讯云联合微众银行、平安科技等二十余家金融机构和科技企业共同发起、成立了金融区块链合作联盟，在国内率先尝试探索、研发和实现适用于金融机构的金融联盟区块链，并基于腾讯云的基础能力，与微众银行联合发布了国内第一个面向金融业的联盟链云服务，让跨金融机构的交易更迅速、成本更低。

早在 2016 年 5 月，华为就加入了由腾讯云发起的金融区块链合作联盟，同年 10 月，华为正式加入 HyperLedger 项目，成为当时加入区块链科技联盟的最大智能手机供应商之一，在两个热度很高的子项目 Fabric 和 STL 中持续做出技术和代码贡献，同时被社区授予 Maintainer 职位，也是这两个子项目中唯一来自亚洲的维护者。

世界经济论坛调查报告预测，到 2025 年，在全球 GDP 中有 10% 的相关信息将用区块链技术保存。也许在未来的几年中，作为编织这一金融基础设施的基本平台，企业级区块链的基础建设将变得十分重要。

2.1.2 Blockchain as a Service

区块链是比特币的底层技术，不仅应用于比特币，还应用于其他不同的行业，因此，“区块链+”将会对众多行业产生重大影响，甚至是颠覆性的变革。由于区块链技术的开发、

研究与测试工作涉及多个系统，因此时间与资金等成本问题将阻碍区块链技术的突破。如果利用云平台搭建测试环境，上述问题则将迎刃而解；同时，区块链所具有的分布式存储特性也将为云服务本身带来深刻的变革。两项技术的融合，将加速区块链技术的成熟，推动区块链从金融业向更多的领域拓展，而两者的结合也构建了一个新的市场——BaaS 市场（Blockchain as a Service，区块链即服务）。

BaaS 指的是为银行、政府部门、金融机构及基于区块链技术研发产品和服务的企业提供区块链技术咨询和定制私有区块链解决方案的一种服务。BaaS 依托公有云平台提供服务，提供基于区块链的搜索查询、任务提交等一系列 API 接口。众多科技巨头们纷纷加入这一前沿行列，目前在这方面，微软、IBM 是两大巨头。

从 2014 年投身比特币市场以来，微软基于 Azure 云计算平台已经与多家区块链初创公司展开合作，致力于在底层操作系统上对区块链提供技术平台支持。早在 2015 年 11 月，微软就联合纽约创业公司 ConsensYs 推出了基于云的区块链技术平台，该平台旨在帮助金融机构以更低的价格使用与比特币等数字货币相关的加密技术，全球四大会计师事务所目前已是该服务的用户。2016 年 8 月，微软区块链服务正式向 Azure 云平台用户开放，开发者可以在云平台上以最简便、高效的方式创建区块链测试环境。总体来说，用户不需要关注具体的实现方式，只需键入参数即可使用，大大降低了开发成本。目前该服务已经支持 26 种不同的区块链实现，包括 Bitpay、MultiChain 和 Storj 等在内的多家区块链初创公司都已经与微软达成了合作伙伴关系，因此这些公司的用户可以直接在微软 BaaS 中找到对应的模板。从这一点也可以看出微软想要聚合所有区块链实现方式的雄心。

2017 年 3 月，IBM 公司发布了一款名为 IBM Blockchain 的新产品，该产品基于 Fabric 项目，是一种 BaaS 服务，用户可将其用于构建安全的区块链网络。据 IBM 公司透露，Blockchain 区块链每秒能够处理超过 1000 笔交易。IBM 声称他们的区块链产品是以一种高度可审计的方式建立的，可跟踪在网络中发生的所有活动，可以让管理人员在出现错误时进行审计跟踪。此外，IBM Blockchain 平台自称是第一种完全整合的企业级区块链平台，被设计用来加速多机构商业网络的发展、治理和运营。IBM Blockchain 目前已经登录 IBM 的 BlueMix 云计算商店，这个商业区块链应用根据实施规模来进行分级收费。我们目前所知道的正在使用 IBM Blockchain 的客户就包括三菱东京 UFJ 银行、Everledger、马士基集团、北美信贷银行和沃尔玛。

云服务市场份额最大的亚马逊肯定也不甘落后。2016 年 5 月，亚马逊选择与数字货币集团（DCG）进行合作，为企业提供一种 BaaS 的试验环境。在合作过程中，AWS 会为这些项目提供专用的云基础设施及技术支持。包括 DCG 在内的一些公司已与该平台进行合

作，探索、测试及部署区块链应用程序。2016 年 9 月，Google 也宣布正式进入 PaaS 市场，为银行提供区块链测试服务，并成功将全球咨询集团纳入自己的区块链技术平台合作伙伴计划中，很显然，除了微软、亚马逊及 IBM，Google 也已经准备好把 BaaS 平台变成一个重要的盈利来源。

2018 年，华为也发布了 BaaS 服务产品——BCS，主打公益、金融保险、IOT 等业务领域，如图 2-1 所示。在区块链领域，华为云将以 BCS 为核心，打造云服务+网络+芯片/终端三位一体的区块链架构，为构建端到端的可信社会体系而努力。

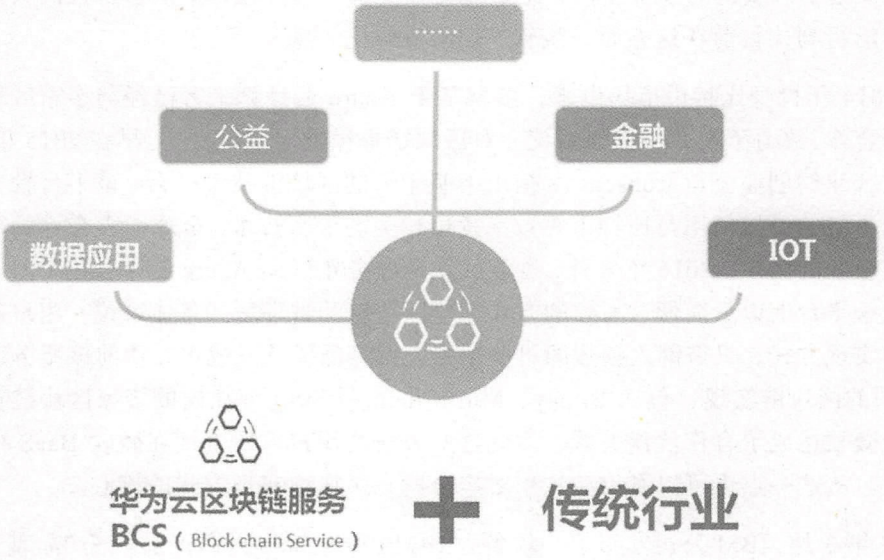


图2-1

2.1.3 区块链的应用场景

目前，区块链的应用已从单一的数字货币应用如比特币，延伸到经济社会的各个领域，比较典型的领域有：金融服务、供应链管理、智能制造、社会公益和可信计算。其中，金融服务为区块链目前最为成熟也最为热门的应用领域，同时，有更多的行业或领域在探索“区块链+”的可能性。全球的区块链技术正以惊人的速度发展，大量的应用正在加速展开。预计在 2020 年，区块链产业将产生 800 ~ 1100 亿美元的收益，而站在最前方的互联网金融有望率先获益。

1. 在金融服务领域中的应用

金融服务行业的典型应用场景之一：支付。在支付领域，区块链技术的应用有助于降低金融机构间的对账成本及有争议解决的成本，从而显著提高支付业务的处理速度及效率，这在跨境支付方面的作用尤为明显，比如瑞波币。另外，区块链技术为支付领域所带来的成本和效率优势，使得金融机构能够处理以往因成本因素而被视为不现实的小额跨境支付，有助于普惠金融的实现。

金融服务行业的典型应用场景之二：资产数字化。数字资产是指企业拥有或控制的以电子数据的形式存在的，在日常活动中持有的以备出售或处于生产过程中的非货币性资产。数字空间的无限可扩展性、无限可复制性、多维可塑造性，可能意味着其中蕴藏着待开发的海量财富，这些财富的表现形式就是数字资产。除数字货币、数字股票、数字债券外，数字资产的范围要大得多，包括所有数字化的资产，比如专利、版权、创意和信用等知识文化资产。数字化的资产很容易被整合到区块链中，成为链上的数字资产，使得资产所有者无须通过各种中介机构就能直接发起交易。同时，任意类型的贵重资产都可以使用区块链技术更好地进行追踪，而记录在区块链中的信息历史记录也无法消除或者篡改。在 2016 年年初，迪拜全球区块链委员会公布了 7 个区块链试行项目，而钻石真伪验证项目就是其中之一。人们希望能够引入一个不可变账本的数字证书，解决金伯利证书造假的难题。

2. 在供应链管理领域中的应用

对于制造业公司而言，如果说从 0 到 1 靠的是研发，那么从 1 到 N 靠的就是供应链，好的供应链管理能够降低产品的运营成本，保证出货速度，是企业的核心竞争力。比如在苹果公司的背后就有一套卓越的供应链管理体系在支撑，而 CEO 库克是著名的供应链专家。随着全球化分工的日益深化，现代企业的供应链在不断延长，出现零碎化、复杂化、地理分散化等特点，给供应链管理带来了很大的挑战。比如一架波音客机就有超过 600 万个零部件，其中 90%由外包供应商制造，这些供应商的数量很多且分布在全球各地。根据 IBM 商业价值研究院发布的报告，这种现状导致了供应链的不透明及信息的不对称，并导致摩擦成本高昂，而区块链这种“信任机器”能够提高核心企业对供应链的掌控力，使产品高效生产、准确溯源成为现实，使商品的信息、物流、资金流都被记录在链上，结合物联网技术，创造出新的商业模式。供应链管理和供应链金融，由于市场规模足够大，满足多信任主体、多方协作、中低频交易、商业逻辑完备等特点，天然是区块链的用武之地，因此备受瞩目。

区块链能够围绕核心企业搭建一条包括制造商、供应商、分销商、零售商、物流公司、终端用户等在内的联盟链，将资金流、信息流、货物流都记录在链上，不可篡改。值得一提的是，货物流上链可以结合物联网技术，简化协同工作。这样一来，区块链就能够实时记录并共享供应链各环节的最新进展，核心企业得以穿透式地实现对供应链的掌控，及时了解订单的生产、质量、运输等情况，将供应链透明化、可视化。透明化的实时管理能够降低企业的库存成本，给予企业应对突发事件的即时支持，也为审计提供了便利。在实时区块链监控领域，物流方面的应用是典型场景。区块链初创公司 BITSE 就和全球最大的物流公司之一德讯物流建立了合作，帮助实时监控货物流。IBM 和世界航运巨头马士基合作，建立起货运公司、代理商、港口及海关之间的联盟链，帮助记录其全球数千万个船运集装箱的情况，预计在大规模应用之后能够为海运业节省数十亿美元的成本。

3. 在智能制造领域中的应用

在智能制造领域，区块链技术也有很好的应用前景。相关研究认为，区块链相较传统的互联网技术的突出优势，在于能明显地减少欺诈、降低成本及提高效率。而这三点的典型优势也正是智能制造的关注点。基于区块链可信的特点，在实施智能制造的过程中可省去如供应商背景调查、产品质量入货检测等基于不可信特点的多余工作；另外，区块链本身具有去中介化的特点，在传统的“互联网+”阶段虽然已经极大地实现了信息透明，但基于电商的大型平台在实质上仍然是一种平台，是一个中介性质的单位，通过导入区块链技术，便可再次缩减诸如电商这样的中间环节，进一步降低实施智能制造的成本。此外，基于区块链的信息透明特点，企业能够在市场上采用最具成本优势的方案。区块链特有的 P2P 特性，可以使智能制造中的各种请求不必从中心系统一层一层地向外传递；另外，区块链减少系统流通环节的特点，本身就是提高工作效率的代表。

除以上三个特点，区块链与智能制造的另一个契合点在于工业物联网的实施。区块链在智能制造领域的渗透，将影响现有工业云企业的布局。基于去中心化的特点，工业云平台之间的品牌差异将逐渐弱化，不同品牌之间或会加速主动兼容；此外，公有云、私有云的概念将逐渐模糊，区块链技术本身集信息透明与隐私保护于一身，突破了公有云、私有云需求环境不同的障碍。

4. 在社会公益领域中的应用

“区块链 + 公益”也是区块链天生适合的应用场景之一。传统公益机构的探路者正积极投入一场改变公益生态的新实验中。蚂蚁金服旗下支付宝爱心捐赠平台目前已全面引入

区块链技术，并向公益机构开放，签约机构经审核后均可自助发布基于区块链的公益项目。壹基金和中国红十字基金会（简称“红基会”）率先提交申请，红基会的首个区块链公益项目“和再障说分手”已顺利上线并实现实时账目公示。区块链是移动互联网的热词，简单来说是一项“不可篡改的数字账本”技术。区块链用于公益领域，除了因为对技术服务生活的想象力，更因为它具有公开、透明的特点。2017 年 7 月，蚂蚁金服与中华社会救助基金会小规模试水区块链公益时，有评论认为，这项技术将有助于解决公益财务透明的痛点。

区块链的技术特点使得其在缺乏信任的场景下有着极大的用武之地。相比而言，网络募捐平台上线的公益项目，无论是捐款人还是平台，通常都只能追踪善款进入基金会账户，但对于每一笔钱究竟何时以何种方式拨付给了哪位受益人，则无法实时监督，只能靠公益机构人工上传项目图文反馈。而区块链技术通过非银行支付机构对资金流向的实时公示，能让公众更直观地了解公益项目的执行方式和流程，有望解决善款公示“最后一公里”的问题。

5. 在可信计算领域中的应用

除了以上与具体业务相关的应用领域，区块链技术的另一个基础的、通用的应用领域是可信计算（Trusted Computing）。可信计算原本是硬件层面的技术，是一种基于硬件安全模块支持的计算，但区块链现在也被视为一种可信计算。如果说 Blockchain 实现了可信记账、可信数据存储，那么，支持智能合约与共识机制的区块链平台就可以被看作一个完整的“可信计算”环境，在这样一个环境中，应用是以智能合约方式执行在安全的沙箱中的，应用的身份是可以通过数字证书识别的，所有的网络数据都是通过安全、可信、可识别身份的通道发送给各个应用的；同时，应用的数据又是被可信存储的。于是区块链又有了下面这种新的定义：

区块链是一种基于可靠数据的通过智能合约执行去中心化的可信计算任务。

当前已经有不少公司在利用区块链技术打造可信计算平台。亦来云（Elastos）宣称要利用区块链技术实现可信记账+可信计算+可信应用环境，希望实现一个安全可信的 App 运行环境。腾讯的区块链方案 TrustSQL 首次用了 Trust 这个词，以强调可信计算，希望与合作伙伴共同推动可信互联网的发展，打造区块链的共赢生态。华为云的区块链服务 BCS 产品，也声称致力于通过区块链技术帮助企业解决数据流转过程中的可信性难题，携手企业共建可信社会。此外，基于可信计算的安全初创公司安全牛也在自己的产品引擎中引入

了一个小型的区块链，同样基于可信计算，目的是解决投票节点的数量问题，确保投票节点的可信性。

目前，我国拥有区块链（创业）企业超过 100 家，仅次于美国，位列全球第二，区块链人才招聘也是一个热点，有不少公司给出了年薪 60 万元人民币以上的 Offer。万达在 2017 年大规模裁撤电商部门，却在 2018 年 5 月成立了区块链子公司，开始涉足区块链领域。虽然区块链技术被各界纷纷看好，但实际上区块链领域仍处于“开荒”阶段，还未形成统一的规范和技术标准。比较好的一点是，区块链技术从一开始就建立在开源的基础上，不管是作为公有链的比特币、以太坊，还是作为企业链的 Hyperledger，对于它们的源码，任何人都可以拿来研究和“高仿”，因此区块链技术几乎不存在巨头垄断的问题，也非常适合初创企业。

2.2 区块链的底层技术与架构

2.2.1 P2P 网络

P2P 网络（Peer to Peer Network）是区块链系统普遍采用的另一个底层技术。为了理解 P2P 网络，我们首先来看看常规的 Master/Slave（或 Server/Client）网络，这是分布式系统普遍采用的一种网络结构，比如我们熟悉的 WWW 采用了典型的 Master/Client 网络，MySQL 的主从复制集群也是 Master/ Slave 网络。在 Master/ Slave 网络拓扑中，只有一个 Master 节点是“领导”节点，其他节点都是“被领导”节点，整个集群的正常运行完全依赖于 Master 节点，如果 Master 节点宕机，则整个集群无法提供服务，并且由于 Master 节点只能有一个，因此集群规模的水平扩展能力也很有限，在集群规模扩大以后，集群能提升的性能及可靠度往往会大打折扣。此外，在 Master 节点无法恢复的情况下，从已有的集群中挑选另一个节点作为 Master 节点来恢复集群，也不容易。

P2P 网络也被称为对等网络，是一种新的网络结构思想，与在目前的网络中占据主导地位的 Master/Slave 结构的本质区别是：在整个网络结构中不存在 Master 节点，如图 2-2 所示。

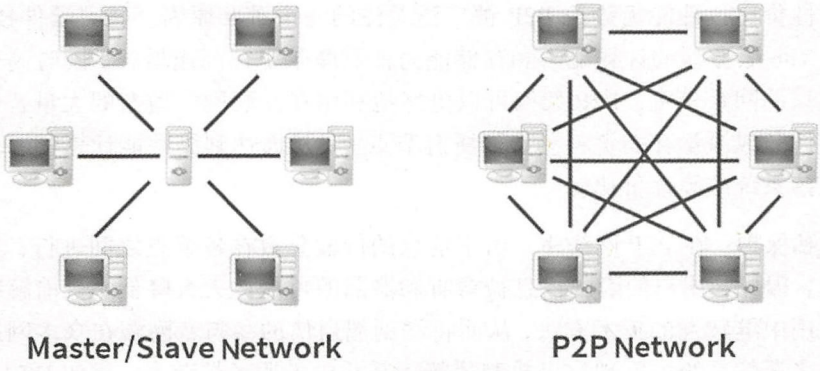


图2-2

P2P 网络的另一个重要特点是改变了互联网现在以大网站为中心的状态，重返“非中心化”，并把权力交还给用户。P2P 网络中的每个节点（Peer）的地位都是对等的，每个节点既充当服务器的角色，为其他节点提供服务，也享用其他节点提供的服务。P2P 网络将传统方式下的服务器的负担分配到网络中的每一节点上，每一节点都承担一定的存储与计算任务，即 P2P 网络中的每个节点都会贡献自己的算力。随着加入 P2P 网络中的节点越来越多，整个集群的能力越来越强，稳定性和服务质量也越来越好，正所谓“众人拾柴火焰高”。

P2P 网络技术的优点主要体现在以下几方面。

（1）去中心化：网络中的资源和服务分散在所有节点上，信息的传输和服务的实现都直接在节点之间进行，无须中间环节和服务器的介入，避免了可能的瓶颈。P2P 的去中心化特点带来了其在可扩展性、健壮性等方面的优势。

（2）可扩展性：P2P 网络通常都是以自组织的方式建立起来的，并允许节点自由地加入和离开集群，因此 P2P 网络的动态扩展性很强。此外，在 P2P 网络中，随着用户（节点）的加入，系统整体拥有的资源和服务能力也在同步扩充，因此，我们在理论上可以认为 P2P 网络的可扩展性几乎是无限的。这种扩展能力带来了很多好处，例如，基于 P2P 技术实现的软件下载，随着用户量的增加，系统提供的资源越来越多，下载速度会变得更快，而在传统的基于 FTP 的文件下载方式中，随着用户量的增加，下载速度会越来越慢。

（3）健壮性：由于服务是分散在各个节点之间进行的，所以在部分节点或网络遭到破坏时对其他部分的影响很小，在部分节点失效时，P2P 网络能够自动调整整体拓扑，确保集群里绝大多数节点之间的连通性，因此 P2P 架构天生具有耐攻击、高容错性的优点。

（4）高性价比：性能优势是 P2P 被广泛关注的一个重要原因。随着硬件技术的发展，个人 PC 与 X86 服务器的计算能力和存储能力依照摩尔定律高速增长，同时高速以太网提供了越来越强的网络带宽，P2P 架构可以更好地利用在互联网中散布的大量普通节点，将复杂的计算任务或海量存储资料分布到所有节点上，从而达到高性能计算和海量存储的目的，并且保持系统的高性价比。

（5）隐私保护：在 P2P 网络中，由于信息的传输分散在各节点之间进行，无须经过某个集中环节，因此，用户的隐私信息被窃听和泄漏的可能性大大降低。目前解决网络隐私问题主要采用中继转发的技术方法，从而将端到端通信的参与者隐藏在众多网络实体之中。在传统的匿名通信系统中实现这一机制依赖于某些中继服务器节点，比如 HTTP 代理服务器在一定程度上保护了用户的隐私。在 P2P 中，所有节点都可以提供中继转发的功能，因而大大提高了匿名通信的灵活性和可靠性，能够为用户提供更好的隐私保护。

正因为 P2P 网络具有这些独特的优势，并且天然符合数字货币去中心化的思想，因此毫无疑问地成为数字货币与区块链系统的重要基石之一，而后者的爆发也让曾经小众的 P2P 技术再次进入大众的视野，正所谓“旧时王谢堂前燕，飞入寻常百姓家”。

1. P2P 网络的核心机制

接下来，我们探讨一下 P2P 网络中的一个核心问题：去中心化的 P2P 网络是如何组网的？为了便于理解，我们从熟悉的 BT 下载开始。要想通过 BT 下载一个文件，首先需要一个种子文件 Torrent，在种子文件中包含至少一个 Tracker（一台服务器地址）信息和文件的分割记录信息。有了 Torrent 文件，BT 客户端软件便能在里面找出 Tracker 地址并建立连接，请求下载此文件，然后我们就正式加入 P2P 的下载网络，Tracker 会返回当前正在下载此文件的其他节点的地址信息，随后，BT 客户端软件会直接跟这些节点建立连接并开始数据传输。在这种情况下，Tracker 成为加入和建立 P2P 网络的关键一环，如果 Tracker 服务器被关闭，我们就无法加入 P2P 的下载网络。为了解决这个问题，DHT（分布式哈希表）技术应运而生，DHT 的出现使得没有 Tracker 也能进行 P2P 网络下载。用过电驴（eMule）的人会发现，在电驴里面有一个选项，可以允许选择 KAD（DHT 的一种）网络进行搜索。

P2P 网络一开始主要用于解决海量文件的共享与下载问题，因此，这些文件被存储在网络的某个（某些）节点中，而网络节点随时会加入或离开，并且节点数量众多，可能有数千万。在这种情况下如何高效、准确地定位某个资源对象（文件）的存储路径，并且确保在网络节点不断发生变动时，这种定位服务也稳定可靠？为了解决这个问题，DHT 模型

应运而生。在 DHT 模型中，网络中的每个节点都被赋予全局唯一的标识——Node ID，Node ID 是对节点的名称或地址信息做哈希运算后产生的一个数字，通常是 128 或 160 位。一个新节点通过在 Bootstrap 配置文件中记录的 Node 地址列表连接到 P2P 网络，随后获取路由信息并计算最靠近本节点的节点列表（通过 A Node ID XOR B Node ID 得到节点之间的距离），然后与这些邻居节点建立长连接，至此，新节点成为 P2P 网络中的一员。然后，此节点便可以发送自己磁盘上存在的文件到 P2P 网络中了，具体过程如下。

（1）这些资源文件先按照名称（关键字）做哈希运算，以得到资源的 ID——Object ID，这里所采用的哈希算法与计算 Node ID 的算法相同，然后寻找 P2P 网络中 Node ID 值最接近此 Object ID 的 N 个 Node（通过 Object ID XOR Node ID 运算得到距离），并向这 N 个 Node 广播新资源信息，这 N 个节点分别生成并保存对应的资源路由信息，这样，即使后面有一个节点宕机，在其他 N-1 个节点上仍能查询到此资源的路由信息。

（2）接下来，如果某个 P2P 节点要查询某个资源，则首先计算此资源的 Object ID，然后计算此 Object ID 的 N 个 Node（通过 Node ID XOR Object ID 运算得到距离），并向这些 Node 发送资源查询信息，如果被查询的节点有这个资源的路由信息，就返回给客户端，否则返回离资源更近的 Node 列表给客户端，直到查询到资源提供者的信息。如果没有查到信息，且没有更近的 Node，就说明资源不存在；如果找到 Node 信息，就向这个 Node 请求资源。

DHT 实际上是一个由广域范围内大量的节点共同维护的一个巨大的分布式散列表，每个节点只负责一小部分资源路由和数据存储，这样即使有节点不断加入或者离开，对整个网络的影响也都很小，于是 DHT 可以扩展到非常庞大的节点规模。

DHT 具有以下性质。

- 离散型（Autonomy and Decentralization）：构成系统的节点是对等的，没有中央控制机制进行协调。
- 伸缩性（Scalability）：不论系统有多少节点，都要求高效工作。
- 容错性（Fault Tolerance）：不断有节点加入和离开，不会影响整个系统的工作。

DHT 是一个模型，而实现此模型的技术或算法有很多种，常见的有 Chord、Pastry、Kademlia 等。其中，我们熟知的 BT 及 BT 的衍生派（Mainline、Btspilits、Btcomet、uTorrent 等），以及 eMule 和 eMule 各类 Mods（verycd、easy emules、xtreme 等）等 P2P 文件分享软件都是基于 Kademlia 算法来实现 DHT 网络的。

2. IPFS 文件系统

近年来，基于 DHT P2P 架构的一个吸引人们眼球的新系统是“星际文件系统”——IPFS（InterPlanetary File System）。IPFS 由来自墨西哥的 Juan Benet 于 2014 年发明。Juan Benet 毕业于斯坦福，在 2015 年参与了大名鼎鼎的 YCombinator 计划（李开复的创新工场的灵感就来源于 YCombinator 的创业孵化商业模式）并成功创立了 Protocol Lab 实验室来研发和推进 IPFS 开源项目。目前 ProtocolLab 实验室已经拥有上百位代码贡献者及十几位核心开发者，因此 IPFS 的开发进度非常好。

IPFS 是一个面向全球的点对点的分布式文件系统，目标是补充（甚至是取代）目前统治互联网的超文本传输协议（HTTP）。HTTP 存在超中心化的问题，使用 HTTP 查找的是资源位置（域名+URL），而使用 IPFS 查找的是资源内容，因此，IPFS 从根本上改变了查找的方式，这是它最重要的特征之一。IPFS 用基于内容的寻址代替传统的基于资源位置的寻址，用户不需要关心存储服务器的位置，不用考虑文件存储的名字和路径，我们在将一个文件存放到 IPFS 节点时，系统基于其内容计算出唯一的加密哈希值并返回给我们作为文件的唯一标识，其背后采用 DHT 技术来解决文件定位问题。此外，IPFS 文件系统基本没有存储上的限制，具体来说，大文件会被切分成小的数据块并分别存储到 P2P 网络的不同节点，在下载时可以从多个节点同时读取。IPFS 不要求每一个节点都存储所有的内容，节点的所有者可以自由选择想要维持的数据。这就像书签一样，除了备份自己的网站，还自愿为关注的其他内容提供服务。由于 IPFS 采用了细粒度的分布式的 P2P 网络，因而可以很好地适应内容分发网络（CDN）的要求，并且可以很好地共享各类数据，包括图像、视频流、静态网站甚至分布式数据库。IPFS 的主要优点之一是完全去中心化的数据存储与共享模式，数据将变得更加分散，即使始发节点是离线的，大部分的数据也都是可见的。如果 IPFS 得以普及，节点数量达到一定的规模，则即使每个节点只存放一点内容，所累计的空间、带宽和可靠性也远超 HTTP 能提供的。

到目前为止，已经有不少项目采用了 IPFS 来进行开发，这里重点提一下与区块链相关的案例。Vitalik Buterin 领导的以太坊作为区块链 2.0 的代表，再次引领了区块链技术的发展潮流。2018 年 1 月 1 日，以太坊正式分叉出 ETF（以太雾），这次诞生的 ETF 不是简单的分叉，而是基于以太坊底层技术的技术革新，简单来说是以太坊（ETH）与 IPFS 的结合体，它拥有更强大的计算能力和分布式储存能力。ETF 在纳入 IPFS 文件系统后从根本上解决了区块链目前的最大问题——无法存储大量的数据，因为区块链要求所有节点都存储完整的账本数据。而通过引入 IPFS，我们可以仅仅将 IPFS 中的链接（Link）和时间戳（Timestamps）写入不可更改的区块链上，从而达到使用区块链安全储存数据的目的，

存储更大规模的账本数据。

3. 小结

最后，我们来总结一下 P2P 网络。从某种意义上来说，P2P 网络和人际网络具有一定的相似性，比如与社交群体（如 QQ 群、微信群等）类似。一般来说，每个 P2P 网络都是众多参与者按照共同的兴趣组建起来的一个虚拟组织，在节点之间存在一种假定的相互信任关系，但随着 P2P 网络规模的扩大，这些 P2P 节点在本质上平等自由的动态特性往往与网络服务所需要的信任协作模型产生矛盾。激励作用的缺失使节点之间更多地表现出“贪婪”“抱怨”和“欺诈”的自私行为，因此在 P2P 中预先假设的信任机制实际上非常脆弱，同时，这种信任机制难以在节点之间扩展，导致了全局性信任的缺乏，这会直接影响整个网络的稳定性与可用性。如何用可靠的机制去解决 P2P 网络中的成员信任问题，就成为可信任的 P2P 网络的一个关键问题，而数字证书与安全技术是解决这一问题的关键，下一节会讲解这方面所涉及的知识。

2.2.2　密码学与安全技术

与密码学相关的安全技术在整个信息技术领域都有很重要的地位。如果没有现代密码学和信息安全的研究成果，人类社会根本无法进入信息时代。而区块链技术在底层大量且直接使用了密码学和安全技术的研究成果，比如采用了安全度很高的椭圆加密算法。实际上，密码学和安全领域所涉及的知识体系十分繁杂而且难以理解，本节仅讲解与区块链相关的一些基础知识，包括哈希算法与数字摘要、加密算法、数字签名、数字证书、PKI 体系、Merkle Tree 等内容。

1. 密码学的基础算法

哈希算法是很基础的计算机算法，它能将任意长度的二进制明文串映射为较短的（通常是固定长度的）二进制串——哈希值，哈希值在应用中又常被称为指纹（Fingerprint）或摘要（Digest），并且不同的明文很难被映射为相同的哈希值，我们在编程过程中用到的哈希表就是哈希算法的一个典型案例。目前常见的哈希算法包括 MD5 和 SHA 系列算法。

- MD5（RFC 1321）是 Rivest 于 1991 年对 MD4 改进形成的版本，仍对输入以 512bit 进行分组，其输出是 128bit 的。MD5 比 MD4 更加安全，但过程更加复杂，计算速度要慢一些。MD5 已被证明不具备“强抗碰撞性”。

◎ SHA（Secure Hash Algorithm）并不是一个算法，而是一个哈希函数族。知名的SHA-1 算法在 1995 年面世，它的输出是长度为 160 位的哈希值，抗穷举性更好。SHA-1 在设计时模仿了 MD4 算法，采用了类似的原理。SHA-1 已被证明不具备强抗碰撞性。

MD5 和 SHA-1 已经被破解，一般推荐至少使用 SHA-256 或更安全的算法，对于任意长度（按 bit 计算）的消息，SHA256 都会产生一个 32 字节的数据。在比特币中有几处用到 SHA256 算法：将公钥转换成比特币钱包地址，如图 2-3 所示；在计算 Merkle Tree 时对每笔交易进行了两次 SHA-256；在挖矿时的工作量证明中用到了，计算出区块的 SHA-256 哈希值小于难度值。

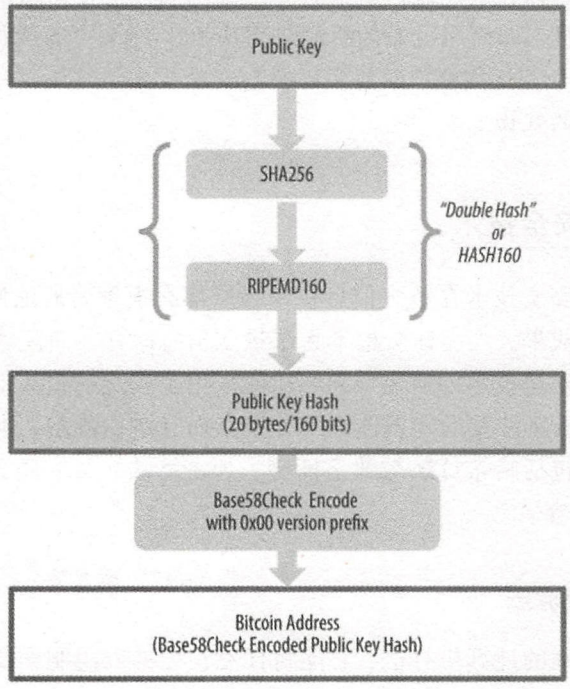


图2-3

哈希算法一般都是计算敏感型的，这意味着计算资源是瓶颈，主频越高的 CPU 运行哈希算法的速度也越快，因此可以通过硬件加速来提升哈希计算的吞吐量，例如采用 FPGA 来计算 MD5 值，可以轻易达到数十 Gbps 的吞吐量。也有一些哈希算法不是计算敏感型的，例如莱特币所采用的 Scrypt 算法在计算过程中需要大量的内存资源，节点不能通过简单地

增加更多的 CPU 来获得哈希性能的提升，这样的哈希算法经常被用在避免算力攻击的场景中。

数字摘要是哈希算法最重要的一个用途，任意长度的一段信息（原文）在经过某种哈希计算后得到的摘要都是相对较短（比如 32 个字节）的一个字符串，原文改变任一字节，都会导致重新计算后的摘要发生变化，因此，经过哈希计算后得到的摘要可以用来解决内容篡改的问题。比如，在通过密钥方式发送消息的时候，消息发送方先选用某种摘要算法（哈希算法）为报文生成一个摘要，收发双方再在摘要的基础上进行报文内容是否篡改的验证。

2. 加密算法的演进

加密算法是密码学的核心技术，在 1976 年以前，所有加密方法都是同一种模式的：

（1）甲方选择某种加密规则（也称密钥）对信息进行加密；

（2）乙方采用同一种规则对信息进行解密。

由于加密和解密采用了同样的密钥，所以被称为对称加密算法（Symmetric-key Algorithm）。常见的对称加密算法有 DES、3DES、TDEA、Blowfish、RC5 和 IDEA 等，相较于 DES 和 3DES 算法而言，AES 算法更快、资源使用效率更高、安全级别也更高，被称为下一代加密标准。对称加密算法的主要优点是加密和解密速度快，加密强度高且算法公开，但其最大的弱点是，甲方必须把密钥告诉乙方，否则无法解密，而如何保存和安全传递密钥，就成了最让人头疼的问题；同时，在对称加密机制下，密钥的分发很困难，在有大量用户的情况下密钥管理复杂，而且无法完成身份认证等功能，不便于应用于网络开放的环境中。

1977 年，三位数学家 Rivest、Shamir 和 Adleman 设计了一种算法（RSA 算法），可以实现非对称加密。RSA 算法基于一个十分简单的数论事实：将两个大素数相乘十分容易，对其乘积进行因式分解却极其困难，因此可以将乘积公开作为加密密钥。从那时直到现在，RSA 算法一直是最广为使用的非对称加密算法（Asymmetric Cryptography，又称公钥加密，即 Public-key Cryptography），是被广泛使用的公钥密码算法，也号称地球上最安全的加密算法。毫不夸张地说，在有计算机网络的地方，就有 RSA 算法。

RSA 首创了公开密钥密码体制，即把在加密解密过程中用到的一个密钥改为公钥与私钥配对的两个密钥，使用不同的加密密钥（公钥）与解密密钥（私钥），并且公开公钥的

一种创新加密机制，从而解决了密钥传输的难题并且奠定了数字证书认证体系的基石。在RSA算法中，私钥一般需要通过随机数算法生成，公钥可以根据私钥生成。公钥一般是公开的，他人可获取；而私钥一般是个人持有的，他人不可获取。以RSA为代表的非对称加密技术有两个典型的应用场景：加密与签名。有意思的是，两者使用的密钥是不同的，可以理解如下：

> 既然是加密，那么肯定不希望别人知道自己的消息，只有自己才能解密，所以公钥负责加密，私钥负责解密；同理，既然是签名，那么肯定不希望别人冒充自己发消息，只有自己才能发布这个签名，所以私钥负责签名，公钥负责验证。

数字签名与在纸质合同上签名来确认合同的内容和证明身份的目的类似，既可以用于证实某数字内容的完整性，又可以确认其来源（或不可抵赖，Non-Repudiation）。数字签名的具体逻辑为：

> A在发送报文信息给B的时候，先选用某种摘要算法（哈希算法）为报文内容生成一个摘要，并使用自己的私钥对摘要操作生成一段“签名”，然后将此签名附在原始报文后面，一同发送给B。B在收到报文后，从中分离出原始报文及签名，使用与A相同的摘要算法计算原始报文的摘要，并使用发送者的公钥及摘要对此签名进行验证，如果验证成功，就表示此签名有效，同时证明原始报文并没有被篡改。

那么问题来了，为什么在数字签名的时候要对报文摘要进行加密，而非对原始报文进行加密？原因很简单，因为非对称加密的算法相对于对称加密来说，性能要低很多，此外，在算法设计上，非对称加密算法对待加密的数据长度有着苛刻的要求，例如，RSA算法要求待加密的数据不得大于53个字节，一般只用于少量数据的加密。

目前比较流行的数字签名的算法有RSA签名算法和椭圆曲线数字签名算法ECDSA（Elliptic Curve Digital Signature Algorithm），后者是使用椭圆曲线密码（ECC）实现的数字签名算法，ECC算法的破译或求解难度基本上是指数级的，这使得ECC算法的单位安全强度高于RSA算法，也就是说，要达到同样的安全强度，ECC算法所需的密钥长度远比RSA算法短，283bit的ECC密码强度大约相当于3072bit的RSA密码强度。ECDSA于1998年被ISO所接受，于1999年成为ANSI标准，并于2000年成为IEEE和NIST标准。

在目前我们所熟知的网银系统中主要使用的是RSA签名（证书）方案，比特币系统则使用了更为安全的ECC签名方案来确保交易的真实性和所有权的认证。在比特币中用户的私钥是一个长度为256 bit的随机数，可通过各种方式生成比特币私钥，本质上就是

在 $1 \sim 2^{256}$ 中选一个数字。比特币客户端软件采用 Secp256k1 ECDSA 标准生成椭圆曲线，使用椭圆生成一个私钥，再从私钥中生成对应的公钥。

3. 数字证书入门

接下来谈谈数字证书。数字证书在本质上可以被理解为用户公钥数据的一种标准化表现格式，目前最广泛使用的数字证书格式是 X509 证书。X509 证书主要由用户公钥和用户标识符组成，此外还包括版本号、证书序列号、签名算法标识、签发者的名称、颁发者的签名、证书有效期等重要信息。如图 2-4 和图 2-5 所示是在谷歌浏览器中打开百度网站后显示的百度证书信息。

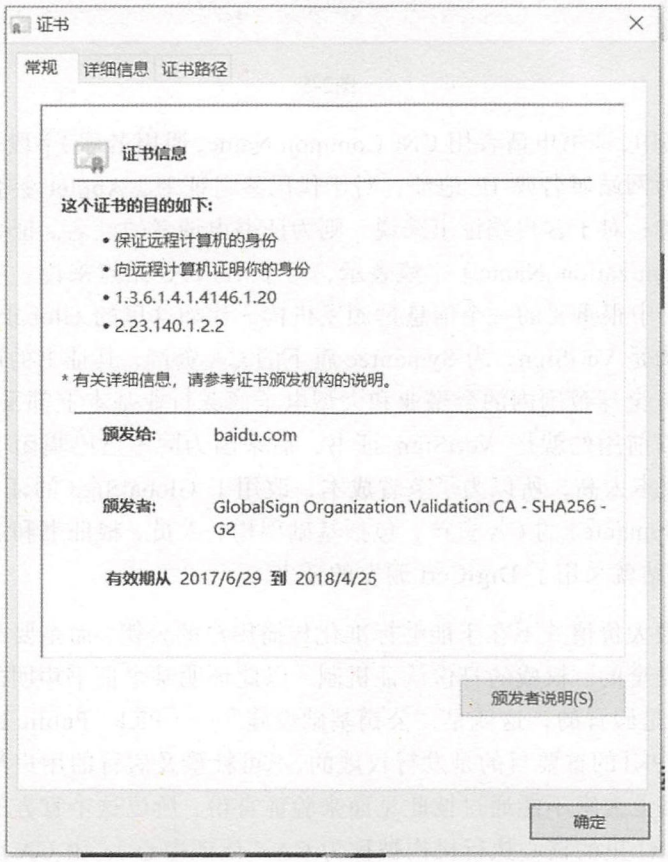


图2-4

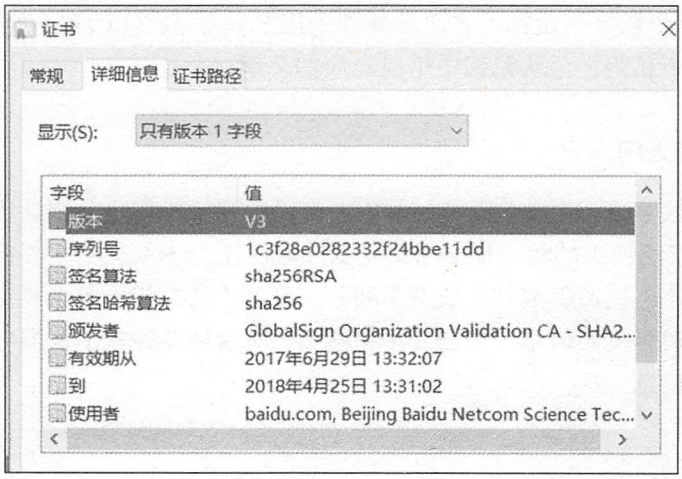


图2-5

在 X509 证书中，证书申请者用 CN(Common Name，通用名称)字段表示，对于 HTTPS 站点来说，一般为网站域名或 IP 地址；对于代码签名证书（Applet 签名证书）来说，则为申请单位的名称；对于客户端证书来说，则为证书申请者的姓名。证书申请者所在的单位名称用 O（Organization Name）字段表示，对于 HTTPS 站点来说一般为网站域名。此外，在 X509 证书中很重要的一个信息是颁发机构，比如这里的 GlobalSign。另外一个知名的证书颁发机构是 VeriSign，为 Symantec 旗下的 CA 资产，其证书的市场更大一些，定位的是高端客户，全球范围内的金融业和大型电子商务行业基本上都采用了 VeriSign 的 SSL 证书。淘宝以前用的就是 VeriSign 证书，后来因为阿里巴巴集团下的域名太多，而 VeriSign 证书的成本太高，所以为了节省成本，改用了 GlobalSign 的证书。2017 年年底，DigiCert 收购了 Symantec 的 CA 资产，包括基础架构、人员、根证书和 PKI 平台。我们所熟知的 GitHub 网站就采用了 DigiCert 颁发的证书。

数字证书的最大价值并不在于能够标准化传播用户的公钥，而是要为基于标准化的数字证书提供一个安全的、权威的身份认证机制，以此证明某个证书声明的主人的确是“真正的主人”，而不是假冒的，这就是“公钥基础设施”——PKI（Public Key Infrastructure）所要实现的目标。PKI 的首要目的是发行权威的、不可杜撰及假冒的用户身份证书——X509 证书，由于在网络上大家不能通过彼此见面来验证身份，所以这个官方的网络身份证书就变得很重要了。PKI 里的核心执行机构被称为 CA（认证中心），由 CA 负责把用户的公钥和用户的其他信息捆绑在一起来生成用户的数字证书，并在网上验证数字证书的真实性，此外，CA 要负责用户证书的黑名单登记和发布。前面提到的 GlobalSign CA、VeriSign CA、

DigiCert CA 都属于公共 CA 机构，可以给全球的任何人与组织机构颁发证书。

前面提到，在 X509 证书上除了有证书颁发机构的名称，还有证书颁发机构的签名信息，那么这个签名的具体作用是什么呢？

答案很简单！有了这个签名后，任何人就都可以通过证书颁发机构的公钥来验证这个签名的合法性了！也就是说，只要得到证书颁发机构的公钥，就能直接验证当前的用户证书是否是这个组织颁发的合法证书，因此，我们的操作系统（含智能手机）、浏览器都内置了主流的公共 CA 机构的根证书。如图 2-6 所示为在笔者的 IE 浏览器中内置的 CA 根证书列表。

图2-6

我们看到，在根证书目录下有一个 CA 沃通根证书，这是我国数字证书颁发机构 WoSign 的证书。因为 WoSign 违规签发证书，所以现在已经被多个浏览器设定为不信任证书机构（被谷歌和火狐封杀），苹果则在 2017 年 1 月向开发者宣布 iOS 11 不再信赖 WoSign CA Free SSL 证书 G2 中级 CA 认证并要求开发者及时更改证书；Google 也在 2017 年 7 月宣布从 Chrome 61 开始不再信任 WoSign 证书。

这里需要说明的是，我们可以自己创建一个证书，把它当作 CA 证书，并且用这个私有的 CA 证书对我们所创建的其他证书进行签名和验证，Fabric 便采用了这样一种常规做法，为了方便用户颁发证书及管理证书，实现了一个单独的 CA Server，并提供了相关的命令行工具，后面会讲到它的用法。

接下来谈谈证书链。通常来说，比较大的证书颁发组织会形成一种类似于“制造商→总代理→一级代理→二级代理→…→经销商→分经销商”的分层销售的组织结构，当我们将自己生成的证书提交给签名商时，签名商会用自己的私钥给我们的签名生成证书，而他们自己的证书又可能来自上一级 CA 证书机构，这样一级级向上，最终到达 CA 根证书，这就是所谓的证书链。我们在验证用户证书的有效性的时候，会一层一层地去寻找颁发者的证书，直到找到最终的根证书，然后通过相应的公钥反过来验证下一级数字签名的正确性。图 2-7 展示了一个由二级 CA 机构签名颁发的用户证书的证书链。

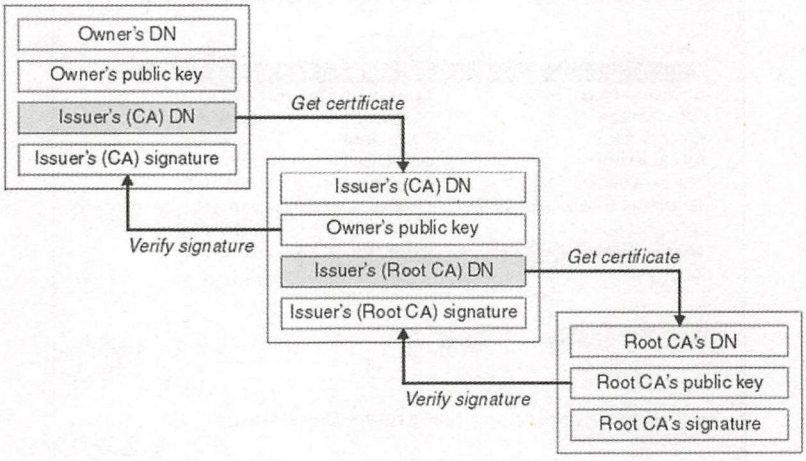


图2-7

4. 通信安全问题

最后，谈谈安全领域的另外一个重要领域——安全传输，涉及 TLS/SSL 及 HTTPS 两部分知识。

我们在早期访问 Web 时采用了 HTTP，该协议在传输数据时采用了明文传输。明文传输带来了以下风险：

◎ 信息窃听风险，第三方可以获取通信内容；

◎ 信息篡改风险，第三方可以篡改通信内容；
◎ 身份冒充风险，第三方可以冒充他人身份参与通信。

为了解决明文传输所带来的风险，网景公司在 1994 年设计了 SSL（Secure Socket Layer）用于 Web 的安全传输协议，这是 SSL 的起源，因为应用广泛，所以已经成为互联网上的事实标准。随后，IETF 于 1999 年将 SSL 标准化，在 SSL 更新到 3.0 时，IETF 将 SSL 3.0 标准化并添加了少数机制（但是几乎和 SSL 3.0 无差异），标准化后的 IETF 被更名为 TLS 1.0（Transport Layer Security），可以说 TLS 就是 SSL 的 3.1 版本。事实上，我们现在用的都是 TLS，只是习惯将其称为 SSL。TLS/SSL 是一种网络加密通道的规范，利用了对称加密、公私钥不对称加密及密钥交换算法，提供了基于数字签名来保证数据完整性的一个安全的通信通道。

TLS 协议采用以下三种机制为信息通信提供安全传输。

◎ 身份认证：通过证书进行认证。
◎ 隐秘性：所有通信都在通过加密后进行传播。
◎ 可靠性：通过校验数据的完整性来维护一个可靠的安全连接。

TLS/SSL 协议工作在 TCP 层之上，在开始加密通信之前，Client 端和 Server 端必须首先建立 TCP 连接并交换参数，这个过程就叫作握手（Handshake）。握手的第一步是 Client 向 Server 发送 Client Hello 消息，在这个消息里包含了一个由 Client 生成的随机数 Random1、客户端支持的加密套件列表（Support Ciphers）和 SSL Version 等信息。随后，Server 会从 Client Hello 传过来的加密套件列表里确定一份加密套件，这个套件决定了后续在加密和生成摘要时具体使用哪些算法，然后也生成一个随机数 Random2，并将这些信息包含在 Server Hello 应答消息里发送给 Client，至此，Client 和 Server 都拥有了两个随机数（Random1+Random2），这两个随机数会在后续生成对称密钥时用到。随后，Server 通过 Server Certificates 消息把自己的数字证书（公钥）发送给 Client，若要进行更为安全的数据通信，则 Server 还可以继续向 Client 发送 Certificate Request 消息请求客户端的证书，以完成双向身份认证。Server 在没有进一步的动作时，会发送 Server Hello Done 的消息给客户端，表示 Server Hello 结束了。Client 在收到 Server 的证书等信息后，会先验证服务端的证书的完整性及在证书上声明的域名（CN 字段）与 Server 的真实域名是否吻合。在通过验证之后，Client 使用之前指定的加密算法生成第三随机数 Random3，并通过 Server 发来的公钥加密 Random3 生成 PreMaster Key，然后发送给 Server，Server 在收到消息后再用自己的私钥解出这个 PreMaster Key，从而得到客户端生成的 Random3。至此，客户端和服务端都拥有 Random1、Random2 和 Random3，两边再根据同样的算法生成一份对

称加密算法所用的密钥，这个过程就是 Client Key Exchange，之后的应用层数据都在使用这个密钥进行对称加密后被传输给对方。我们看到，生成数据加密用的密钥一共需要三个随机数，因此 TLS/SSL 协议非常安全。Fabric 在设计时就考虑了数据传输的安全性，因此网络中的节点可以选择开启 TLS/SSL 安全通道，也可以选择开启常规的 TCP 通道。

在理解 TLS/SSL 协议后，我们再看 HTTPS 协议就非常简单了，HTTPS 相当于 HTTP over SSL，因此我们可以理解为什么提供 HTTPS 服务的站点需要购买和部署数字证书了，同时我们理解为什么在 HTTPS 站点上提交用户名和密码后，不会被中途拦截和嗅探，这是因为 HTTPS 在传输数据的过程中采用了 TLS/SSL 通道加密技术。

5. 小结

我们现在大致明白了数字货币与区块链系统的底层原理：通过 P2P 网络组建一个去中心化的超级网络，任何人都可以加入这个网络中进行交易；为了防止欺诈并确保系统安全可靠地运行，采用了数字证书身份识别、加密签名技术及 TLS/SSL 安全传输技术。那么，接下来又有一个新问题：在大规模的 P2P 网络中，每笔交易与区块数据是如何快速传播到全网每个节点的呢？下一节就来探讨这个问题。

2.2.3 Gossip 协议

在比特币网络中，每笔交易都被打包成一个区块并被广播（Broadcast）到每个节点，之后的每个节点都将区块数据写入本地数字账本中并持久化保存。如果这个网络是一个中心化的网络，那么消息“广播”就是一个容易实现的逻辑，中心节点向每个节点发送报文即可。但比特币网络是一个去中心化的 P2P 网络，每个节点只与周围的几个节点建立连接，在这种情况下，一个消息要实现全网广播则看起来不太简单。直觉告诉我们，一定会有某种算法协议来实现，这就是大名鼎鼎的 Gossip 协议！Gossip 的意思是“流言蜚语”，俗话说得好：好事不出门，坏事传千里！这就是七嘴八舌的“传播威力”。Gossip 协议就是 P2P 网络中很神奇的一个协议，用于消息传播，其传播方式也与病毒传播类似，因此 Gossip 协议有众多别名，例如闲话算法、疫情传播算法和病毒感染算法和谣言传播算法。Gossip 协议并不是一个新事物，之前的泛洪查找、路由算法都在其范畴之内，不同的是 Gossip 协议给这类算法提供了明确的语义、具体的实施方法及收敛性证明。Gossip 协议有原理简单、实现方便、分布式容错等优点，因此成为分布式系统中被广泛使用的基础协议之一。

在 20 世纪 60 年代，耶鲁大学的社会心理学家米尔格伦设计了一个连锁信件实验，他

写了一封信，在信中写下一个波士顿股票经纪人的名字，在信中要求每个收信人都将这封信寄给自己认为比较接近那个股票经纪人的朋友。朋友在收信后照此办理，将这封信随机发送给居住在内布拉斯加州奥马哈的 160 个人。最终，大部分信在经过五六个步骤后都到达该股票经纪人这里，这就是人际关系网络中著名的六度分离理论（Six Degrees of Separation）的最早试验。这个理论告诉我们："你和任一陌生人之间所间隔的人不会超过五个，也就是说，最多通过五个人你就能认识任一陌生人"。这个理论实际上说明了在去中心化的 P2P 网络中，信息传播的速度和距离是我们难以想象的。

在一个有界网络中，每个节点都随机地与其他节点通信，经过一番杂乱无章的通信，即使有的节点因宕机而重启，或有新节点加入，在经过一段时间后，这些节点的状态也会与其他节点达成一致，在最终的某个时刻，网络中所有节点的状态都会达成一致。因此，Gossip 协议归属于最终一致性算法，而不是强一致性算法。Gossip 协议也有明显的缺点，即冗余通信会对网络带宽、CPU 资源造成很大的负载。

很多去中心化的系统都采用了 Gossip 协议，比如服务发现框架 Consul 就用了 Gossip 协议管理集群成员关系并实现消息广播，Consul 利用了两个不同的 Gossip Pool，我们分别把它们称为局域网池（LAN Pool）或广域网池（WAN Pool）。每个 Consul 数据中心都有一个包含所有成员的 LAN Gossip Pool。LAN Pool 有这些目的：首先，成员关系允许 Client 自动发现 Server 节点，减少所需的配置量；然后，分布式故障检测机制使得故障检测的工作可以被分配到某几个 Server 节点执行，而不是集中在整个集群的一个节点上；最后，允许可靠和快速地进行事件广播，比如 Leader 选举。Consul 的 WAN Pool 是全局唯一的，无论属于哪个数据中心，所有 Server 都应该加入 WAN Pool 中，由 WAN Pool 提供成员信息让 Server 节点可执行跨数据中心的请求。此外，在 Akka 集群中也采用了 Gossip 协议，例如集群当前的状态就是通过 Gossip 算法广播给集群中的每个节点，确保每个节点都看到当前最新的（状态）版本。但真正让 Gossip 协议名声大噪的就要数 Cassandra 了。Cassandra 集群是一个 P2P 集群，没有中心节点，采用 Gossip 协议维护集群的状态。Cassandra 在启动时会启动 Gossip 服务，Gossip 服务会运行一个 GossipTask 任务，这个任务会周期性地与其他节点进行通信。通过 Gossip 协议，每个节点都能知道集群中的成员列表及它们的最新状态，这使得 Cassandra 集群中的任一节点都可以完成任意 Key 的路由，任一节点不可用都不会造成灾难性的后果。Cassandra 主要采用了 Gossip 协议来实现下面这些重要功能。

- 失败检测。
- 动态负载均衡。

◎ 去中心化的弹性扩展。

本节最后，我们谈谈 Gossip 协议的实现方式。Gossip 协议的实现方式可以分为以下两种。

◎ anti-entropy：只要数据不同步，就开始同步数据。
◎ rumor mongering：间隔固定的时间同步数据。

在比特币网络中采用了 anti-entropy 的实现方式，即每当一个节点发现自己账本中的区块链高度低于其他节点时，就开始通过拉取的方式实现相邻节点之间的账本同步。在 Fabric 集群中也采用了 Gossip 协议来实现区块数据的广播和账本同步逻辑，在后面的章节中会详细讲解。

2.3 区块链平台架构

2.3.1 区块链平台的常规架构

区块链平台的常规架构如图 2-8 所示，由下往上可以分为以下几部分。

◎ 数字账本，存储区块链数据。
◎ P2P 网络，在各个节点之间建立高效安全的数据传输通道。
◎ 成员身份识别系统，通常通过私钥公约技术来识别与验证区块链的节点成员。
◎ 共识机制，为在分布式集群中维护区块链账本的一致性的某种算法实现。
◎ 智能合约引擎，为执行智能合约代码的模块，通常采用虚拟机技术来实现。
◎ 服务接口，面向业务系统提供账本数据的编程接口。
◎ 区块链业务系统，指某个具体的区块链应用，例如各种虚拟数字货币和基于区块链技术的专项业务系统。

设计得比较好的区块链平台都会考虑各个模块的扩展性与可替换性，比如 IBM 的 Fabric 平台，其智能合约引擎可以支持多种编程语言的开发，共识机制模块有多种共识算法插件可选，数字账本模块有多种存储机制可选，例如 LevelDB、CouchDB 及 MySQL。

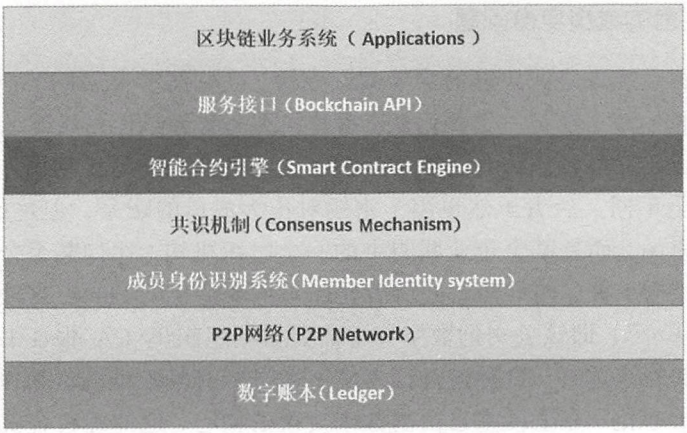


图2-8

1. 区块链平台的网络机制

区块链系统由于是去中心化的分布式系统，因此没有采用常规的 Master/Slave 模型，而是采用了基于 P2P 的网络通信方式。P2P 网络的节点之间相互连接、协同，每个节点在对外提供服务的同时，也采用网络中其他节点提供的服务，每个节点既是服务端又是客户端。P2P 网络不仅去除了中心化带来的风险（中心化可能作恶），还可以提高传输的效率，被广泛使用的 BT 下载就是基于 P2P 网络提升大文件的下载速度的。P2P 网络中的客户端节点维持了一个长期稳定运行的“种子节点”列表（其实和 BT 下载的种子文件道理是一样的），可以通过种子节点来快速发现网络中的其他节点。比特币的 P2P 网络具有以下特点。

- 特定的协议（基于 TCP 端口 8333）。
- 以随意的拓扑结构连成网络。
- 所有节点平等。
- 新节点随时可加入。
- 无响应的节点在 3 小时后会被遗忘。

Fabric 在其 P2P 网络通信模块中采用了 Gossip 协议，以加快 Block 区块在整个集群中的广播速度。共识机制的问题在前面的章节中讲了很多，这里需要说明一下，共识机制并非区块链的本质特征，比如 Fabric 1.0 默认的共识机制为没有共识，即只有单一的节点（Orderer）负责对所有交易排序、打包、验证和分发，因此的确没有共识可言。

2. 区块链平台的成员身份问题

成员身份识别系统也是区块链的重要基础模块，也可称之为“账户系统”。我们知道，比特币里的每个账号其实都是一个私钥&公钥对，不需要经过任何组织（比如 CA 机构）的签名和身份验证，就可以作为合法账号使用，比如挖矿、交易比特币。这与企业联盟链 Fabric 的做法完全不同，后者虽然也用了密钥对作为成员的账号，但这里的成员（节点）并非面向个人用户的，而是某个企业联盟中的一个组织机构，比如某个公司或者某个集团的下属分公司（机构）等。如果某个公司需要加入一个特定的 Fabric 区块链网络中，就必须获取“组织的认可”，即该公司的数字证书需要得到组织里 CA 根证书的签名认可，这也是企业联盟链的特点之一。这种通过证书签名来进行身份验证的区块链，从本质上来说已经偏离了完全中心化、完全民主自由的数字货币的初心，但的确符合我们既定的商业准则和企业行为，同时，由于加入企业链的每个企业的身份都是严格验证的，所以不靠谱的敌对的企业不会被纳入本组织中，因此在企业区块链中出现所谓的“拜占庭将军”的可能性几乎为零，这就是为什么 Fabric 1.0 在正式发布的时候取消了拜占庭共识算法，共识服务默认为 Solo（没有共识）或可选的 Kafka（采用消息队列来实现“伪共识”）。

3. 区块链平台的智能合约实现方式

与以太坊需要自己动手开发语言及虚拟机的思路不同，Fabric 选择采用现有的 Docker 容器技术来支持智能合约功能。Fabric 的智能合约在理论上可以用任何语言来编写，这一点对开发者相当友好，他们无须学习新的语言，可以复用现有的业务代码和丰富的开发库，并使用自己熟悉的开发工具，更容易实现智能合约的版本控制功能。相对地，采用 Docker 的智能合约架构也有自身的问题：首先，构架、部署及调用复杂，也难以对智能合约的执行流程进行控制并限定其所能执行的功能；其次，无法对合约运行所消耗的计算资源进行精确评估，也无法在移动设备上执行智能合约，即移动设备无法成为 Fabric 的区块链网络中的节点，但这不影响 Fabric 的定位和出发点，因为它本来就是企业级联盟链，而非公有链。

关于智能合约，这里需要说明的是：智能合约虽然是用户开发的一段代码，但它们是嵌入在区块链平台（而非用户自己的业务系统）中执行的，因此，智能合约可以被认为是区块链核心的一个“扩展机制”。我们开发的区块链应用只能通过编写智能合约来操作区块链数据，因此学习和掌握一种区块链平台的智能合约开发技能，对于区块链应用开发来说至关重要。

最后说说区块链平台中的服务接口（Blockchain API），以 Fabric 为例，它所提供的主要 API 接口如下。

- 调用或触发某个智能合约（特定交易）的执行。
- 查询区块链信息。
- 读取或写入区块的内容。
- 查询交易信息。
- 监听交易事件。

在后面的章节中，我们会对这些接口的使用给出详细的例子。

4. 腾讯的企业级区块链产品架构

本章最后，我们再来分析一下腾讯公开的企业级区块链产品架构，如图 2-9 所示是在其官方白皮书里给出的区块链基础框架 TrustSQL。

图2-9

从图 2-9 中可以看到，腾讯区块链基础框架从下往上分为三大层。

（1）区块链底台平台：是区块链系统的核心部分，提供了包括用户管理、基础服务（接口适配、共识管理、网络通信、记录存储等）、智能合约，以及运营监控等。

（2）平台产品服务层：实现了区块链应用开发所需的基础通用服务，例如数字资产、鉴权服务、共享账本、分享经济等。

（3）应用服务层：提供了基于区块链方案的各类区块链场景的服务给最终的用户使用，目标是为用户提供可信、安全、便捷的区块链服务，比如数字票据、贵金属交易、知识产权保护、网络互助、机构清算及公益等。

2.3.2 Fabric 的原理与架构

1. 账本与交易

账本主要是用来记录交易数据的，需要清晰地记录在每笔交易中谁在什么时候支付了谁多少。在传统应用中，我们将这些数据记录在数据库内；而在 Fabric 中，我们将这些数据记录在区块链文件（Blockfile）内，这也意味着在最基本的 Fabric 网络架构中，我们需要有一个 Committer 节点及其所对应的操作的区块链文件。假设由 A 与 B 发起了一笔交易，我们并不关心具体交易的内容，这个交易提案产生的相关交易最终由 Committer 节点写入 Blockfile 中持久化保存。如图 2-10 所示。

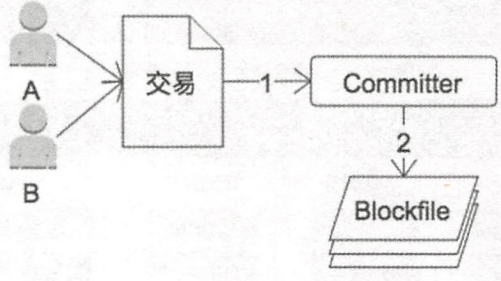


图2-10

那么是不是任一交易提案都能被写入 Blockfile 中呢？当然不是，Committer 在写入交易前需要对交易提案进行验证并签名确认，这叫作背书（Endorsement），Endorser 会先查看交易提案是否合法，以及是否可以正常提交，只有在 Endorser 验证通过的时候，交易提案才会被提交给 Committer 写入 Blockfile，这个流程如图 2-11 所示。

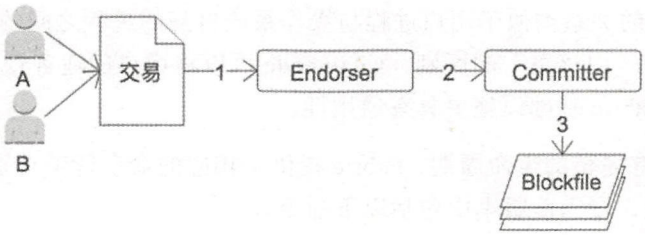


图2-11

2. 交易背书机制

通过对前面章节的学习，我们知道，Fabric 联盟链中的多方组织（Org）都要对在 Fabric 网络中发生的任意交易提案的合法性做出“共同决策”，这是通过背书策略（Endorsement Policy）来实现的。Fabric 允许我们为不同类型的交易提案设置不同的背书策略，Endorser 在节点收到一个新交易提案后，会根据背书策略对应的规则来决定是否放行或拒绝，背书策略的规则可以是一方认可、多数认可或者全部认可。如果我们在安装部署某个 Chaincode 的时候没有指定相关的背书策略，Fabric 就会为我们的交易提案设置一个默认的背书策略，它的规则如下：

> 目标 Chaincode 所在的通道上的任一组织（Org）内的一个成员批准即可放行交易。比如，在我们创建的 Channel1 上有 Org1、Org2 这两个组织，则默认的背书策略的规则为 OR(Org1.member,Org2.member)。因此，我们把交易提案发给 Org1 或 Org2 上的任一 Peer 节点（Endorser 节点），在这个 Peer 节点审批通过并背书签名该交易提案后，交易提案就可以进入下一个环节，最终被提交到 Committer 节点。

如果我们希望第 1 章的转账例子更安全，就可以设置背书策略，策略规则为 AND(Org1.member,Org2.member)，这样一来，所有转账交易的提案就都必须通过两个不同组织内的节点背书才能进入下一个环节。

3. Fabric 的智能合约

通过对前面章节的学习，我们知道，在 Fabric 中一个交易的具体逻辑是在 Chaincode 中运行的，智能合约是用户编写的一段可执行程序，与 Fabric 自身的程序是相互独立的，它们之间通过 gRPC 远程方法调用来实现通信，因此 Chaincode 与 Fabric 网络具有一定的

隔离性，它们之间的关系类似于用户进程与操作系统自身的进程之间的关系，或者简单地说是“用户态”与“内核态”的区别，Chaincode 可以将用户的业务逻辑部分从平台中剥离出来，从而让 Fabric 基础设施更具有通用性。

Chaincode 拥有完整的生命周期，Fabric 提供了相应的命令行工具来实现对 Chaincode 全生命周期的管理，在当前版本中包括以下命令。

- package：将 Chaincode 源码打包成可部署的安装包。
- install：安装打包好的 Chaincode。
- instantiate：初始化安装好的 Chaincode。
- upgrade：升级某个 Chaincode。

Chaincode 在成功安装及实例化后，以一个 Docker 容器的方式处于运行状态，我们就可以通过 Fabric 命令行工具或者 Fabric API 编程调用 Chaincode 的交易接口，后续也能够对 Chaincode 进行升级。在 Fabric 未来的版本中会增加 stop 及 start 命令，用于停止与开启 Chaincode，而不用卸载 Chaincode。

增加了 Chaincode 的交易流程如图 2-12 所示。

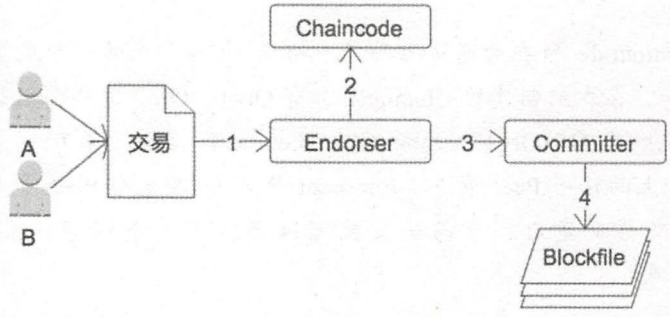


图2-12

除了用户开发的与具体业务相关的 Chaincode（User Chaincode），Fabric 也实现了一些系统级别的 Chaincode（System Chaincode），System Chaincode 与 User Chaincode 有着相同的编程接口，但与后者有以下不同之处。

- System Chaincode 的代码被包含在 Peer 节点的可执行文件中，属于 Fabric。
- System Chaincode 在 Peer 节点的进程内执行，不在隔离的 Docker 容器内运行。
- System Chaincode 不存在生命周期过程，即没有安装、实例化及升级等步骤，只能通过 Peer 节点的二进制文件升级。

- System Chaincode 必须通过一组固定的参数进行注册，不存在对应的 Endorsement Policy。

System Chaincode 实现了一系列系统管理功能，以便于系统集成人员根据需求对它们进行修改与替换，同时减少了 Peer 节点与 User Chaincode 之间 gRPC 通信的代价。Fabric 现有的 System Chaincode 系统如下。

- LSCC（Lifecycle System Chaincode）：参与 User Chaincode 的生命周期管理，User Chaincode 的安装、实例化都会自动触发对 LSCC 智能合约的调用。
- CSCC（Configuration System Chaincode）：参与通道的配置管理。
- QSCC（Query System Chaincode）：提供 Fabric 账本查询的 API 接口，例如获取 Block 与 Transaction 的信息。
- ESCC（Endorsement System Chaincode）：参与交易提案（Transaction Proposal）的背书签名。
- VSCC（Validation System Chaincode）：参与 Transaction 的验证，包括检查背书策略是否满足，以及实现多进程并发访问控制。

与 Transaction 密切相关的两个 System Chaincode 分别是 ESCC 与 VSCC，其中 ESCC 完成了交易提案的背书签名过程，通过 ESCC 背书的交易提案最终会生成 Transaction，并被打包到 Block 中发给 Committer 写入账本。那么，Committer 如何验证 Transaction 数据是否在网络传输的过程中被篡改呢？原来，Committer 在写入之前会调用 VSCC 对 Transaction 数据进行再次验证。VSCC 主要用于验证背书签名是否有效、背书签名是否来自有效的背书节点，以及是否满足背书策略。只有在通过 VSCC 的验证之后，交易数据才会被写入 Blockfile 中持久保存。增加了 System Chaincode 的 Fabric 交易流程如图 2-13 所示。

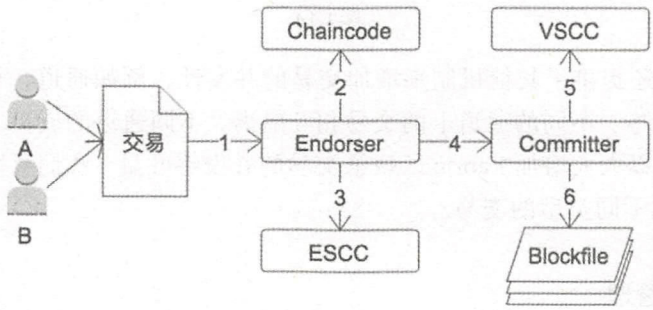


图2-13

4. 高并发交易的设计思路

对于现在的互联网应用来说，高并发的交易已经成为基本需求，只有一个 Endorser 和一个 Committer 很难满足大量的高并发需求，如果只对 Endorser 和 Committer 进行数量上的简单增加，则会导致网络复杂度增加，同时需要解决多个节点之间的数据同步及分布式系统中常见的交易顺序问题。交易记录的顺序直接决定了某笔交易是否有条件发生，而每笔交易都有可能决定之后一系列的交易能否发生，因此对交易的排序对整个系统来说都非常重要，如何解决分布式系统中不同节点发生的事件先后顺序是每个系统在设计时都需要考虑的一个关键问题。Fabric 为了解决这些问题，增加了交易排序节点 Orderer。Orderer 节点并不关心具体的交易内容，只关心交易的顺序和分发对象，经 Endorser 验证通过的所有交易都被统一提交到 Orderer 节点中，由 Orderer 节点排序后提交给 Committer 进行记录。需要注意的是，因为区块链的写入是一种阻塞性操作，所以为了提高系统的处理能力，Orderer 节点允许将一定数量的交易打包成 Block 后再提交给 Committer。如图 2-14 所示是增加了 Orderer 节点的交易流程。

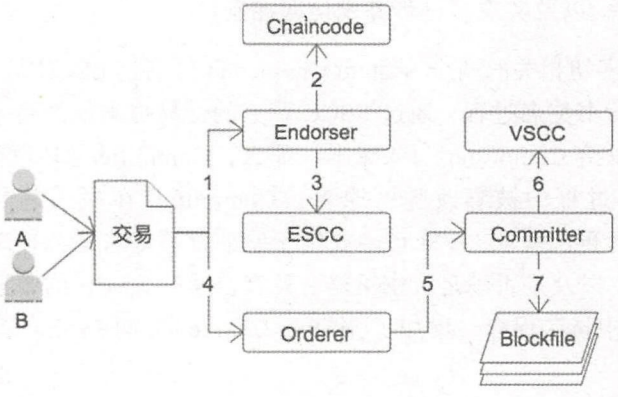


图2-14

当然，Fabric 还提供了其他机制来增加交易的并发性，例如通道，每个通道对应一个逻辑上的区块链账本，不同的通道上的交易相互隔离，不同通道的成员之间的通信也互不可见，因此通道可以大大增加 Fabric 区块链交易的并发吞吐量，我们可以根据业务场景设计多个通道来承载不同类型的交易。

5. 成员身份管理

现在，我们看到 Fabric 网络越来越复杂了，可能存在很多个 Endorser、Orderer、

Committer，那么如何管理这些角色，并保证每个节点之间的通信是合法的、未被篡改的呢？这就需要引入身份验证机制，即之前所提到的 MSP（Membership Service Provider）。在 MSP 机制下，每个节点（Orderer 或 Peer）都拥有自己的证书，每个节点发出去的消息都会用自己的证书进行签名，同时对端节点可以对此消息进行真伪验证。

Fabric 对节点的管理就如同一个私人俱乐部，不允许陌生人进入，陌生人在进入时需要告诉保安："我是 Neo 介绍来的"，如果保安认识 Neo，则允许这个人进入，否则不允许。通过熟人关系产生的信任链在互联网中表现为 CA（Certificate Authority）证书制度，整个系统拥有一个根证书（Root CA），由根证书签发中间 CA 证书或者"收录证书"（Enrollment Certificates），只有持有合法证书的节点提交的交易才能被其他节点认可；除了收录证书，Fabric 还内置了支持传输层加密的 TLS（Transport Layer Security）安全协议，基于传输层证书对交易进行加密，防止在传输过程中对消息进行窃听与篡改。在一般情况下，MSP 证书可以由 cryptogen 工具生成，同时可以由 Fabric-CA 管理。

2.3.3　Fabric 架构总结

至此，我们已经对 Fabric 的整体结构和角色有了更深刻的理解：

> 客户端先将交易提案提交给 Endorser 进行背书，通过背书的交易提案会被提交到 Orderer 节点排序并生成交易记录，随后 Orderer 节点会把一系列交易打包成 Block 提交给 Committer，Committer 会在交易验证后将 Block 写入区块链文件中持久化保存。Fabric 的多组织（Multi -Organization）与多通道（Multi-Channel）也是这个项目的特色之一，多组织主要是指交易在两个或两个以上的组织间进行时，可以由多方组织进行背书，这些组织既可以是公司内部不同的部门，也可以是不同的公司。

如图 2-15 所示是拥有 Org1、Org2 这两个独立组织的 Fabric 网络示意图。

多通道又被称为多链，也是 Fabric 的另一个特性。在 Fabric 网络中，我们可以维护多个通道，每个通道都有自己的成员（Member），每个通道都对应自己的区块链文件，不同的通道在逻辑层面上相互隔离，因此也可以作为区块链多租户的一种实现方式。如图 2-16 所示为拥有 Channel1、Channel2 这两个通道的 Fabric 网络示意图。

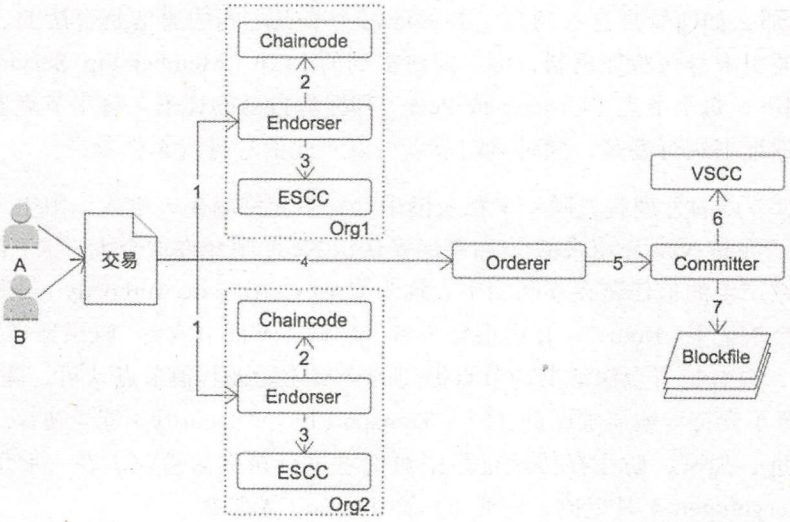


图2-15

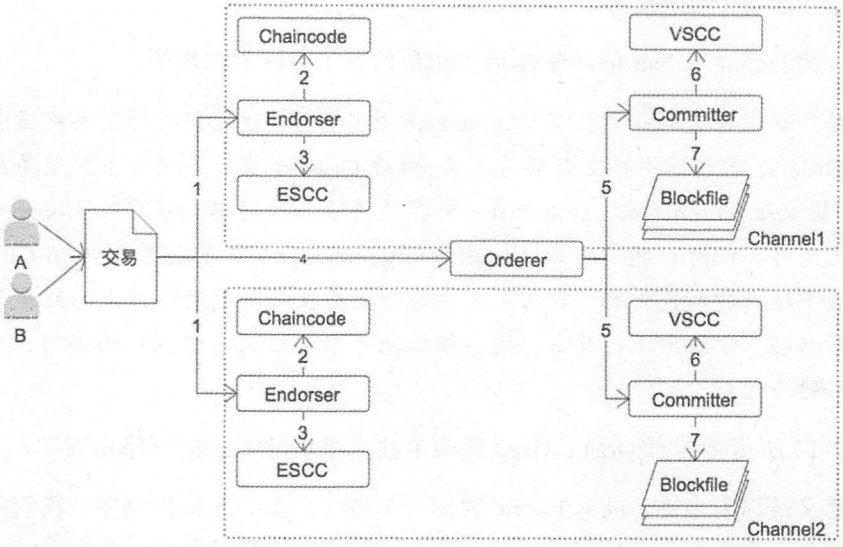


图2-16

通过不同的组织与通道，我们可以组织复杂的 Fabric 网络，以应对各种复杂的业务。在 Fabric 项目中还引用了许多第三方框架与工具，接下来我们根据不同的层面一一进行介绍。

（1）在网络方面，在 Fabric 中最重要的通信框架是 gRPC，在代码中可以看到大量的 protobuf 定义的消息结构及对应的处理。在 gRPC 框架之上通过 Gossip 协议实现了节点之间的通信，满足了在大量的分布式节点之间一对多及数据同步的需求；对于 Orderer 集群来说，集群的排序服务依赖于 Kafka 及 ZooKeeper。

（2）在配置方面通过 viper 工具来管理程序的运行时配置。需要注意的是，viper 支持多种配置方式，包括程序设定值、配置文件、环境变量、命令行参数、远程配置等，因此在使用 viper 时需要注意配置覆盖的优先级顺序：程序设定>命令行参数>环境变量>配置文件>Buffer 键值对>默认参数；在 Fabric 中，大多数配置文件是以 YAML 文件的格式存储的；而在演示环境中，尤其是在 Docker 环境下，大多数配置是由环境变量完成的。

（3）在加密方面，MSP 依赖的证书签名机制及加密算法都由区块链加密算法提供者 BCCSP（Blockchain Crypto Provider）实现，BCCSP 支持基于 PKCS#11 的硬件加密与软件算法加密这两种方式；软件加密主要采用 X.509 的公钥授权标准，在这个标准下支持 ECDSA、AES、RSA 算法生成签名证书，同时支持 SHA2 哈希算法（BCCSP 支持更多的哈希算法，但部分代码暂时写死了 SHA2）。

（4）在持久化方面，Fabric 中的区块链数据是以文件形式存储的，为了加速获取对应的区块数据，区块索引采用 LevelDB 进行存储。为了方便快速地交易验证，账本的状态被存储在 LevelDB 或者 CouchDB 中。

（5）在外部接口方面，Fabric 目前支持 CLI 命令行与 SDK 这两种方式，CLI 主要由 peer 命令支持。SDK 支持以下几种语言与接口。

- Java（参见 GitHub 网站的 hyperledger/fabric-sdk-java 项目）。
- Node.js（参见 GitHub 网站的 hyperledger/fabric-sdk-node 项目）。
- Golang（参见 GitHub 网站的 hyperledger/fabric-sdk-go 项目）。
- Python（参见 GitHub 网站的 hyperledger/fabric-sdk-py 项目）。
- Rest SDK（参见 GitHub 网站的 hyperledger/fabric-sdk-rest 项目）。

我们注意到，Fabric 在 1.0 版本中移除了 Rest 接口，现在这些 Rest 接口在通过封装 Fabric Node.js SDK 后以独立项目的方式提供，以方便更多的人将 Fabric 整合到现有的系统中。接下来，我们通过搭建 Fabric 开发环境、Fabric 网络及进行部分源码解析，来逐步深入地学习 Fabric。

第 3 章
Fabric 安装与调试

3.1 Fabric 源码安装

得益于容器技术的快速发展，很多软件在大多数环境下都可以通过 Docker 直接下载和运行，Fabric 也一样。通过下载官方示例（参见 GitHub 网站的 hyperledger/fabric-samples 项目），我们可以快速建立基于 Docker 的示例网络。这里并不会讲解这些示例，而是希望从搭建编译运行环境开始，建立一个完整的开发调试环境，这便于我们深入了解、修改和定制自己的 Fabric 项目。

在编译环境搭建完成后，我们会运行一个基于二进制程序的 Fabric 网络示例。与之前提到的官方示例不同，这个网络并不是基于 Docker 运行的，而是基于我们编译生成的本地可执行程序运行的。我们通过这个示例，可以建立一个用于快速开发的“编码-构建-运行-调试”本地环境；也可以通过手动运行这个示例的每个步骤和参数，了解 Fabric 网络的原理；还可以根据每个步骤的详细 Log 信息查找对应的源码，建立对源码结构的初步印象。

Fabric 的编译环境较为复杂，依赖项也较多：Git 主要用于管理源码的版本；1.9 版本以上的 Golang 是 Fabric 的主要开发语言；Docker 用于打包部署；Python 用于实现部分测试工具，快速组网用到的 DockerCompose 也依赖于 Python；如果想用开发基于 Node.js 的 Chaincode 或者 SDK，则需要安装 Node.js 环境。下面会从零开始安装这些软件环境。本

章内容以 CentOS 7.4 操作系统为例，和其他操作系统的安装步骤会有一些不同，但大体上类似。需要注意的是，在 Linux 下编译及安装 Fabric 时，需要拥有 root 用户的权限。

3.1.1　基础环境安装

1. 设置 yum 源

请确定 CentOS 系统已安装设置完成，可以访问网络。另外，为了快速、方便地安装，请将 yum 源设置为国内镜像，操作步骤如下。

（1）备份原有的 yum 库设置文件：

```
[root@localhost ~]# mv /etc/yum.repos.d/CentOS-Base.repo \
/etc/yum.repos.d/CentOS-Base.repo.bak
```

在演示代码中使用的是阿里云国内镜像，若有需要，则也可以将其替换成自己需要的地址。

（2）考虑到在 CentOS 最小化安装包中不包含 wget，所以这里使用 curl 下载 yum 库的设置文件：

```
[root@localhost ~]# curl -o /etc/yum.repos.d/CentOS-Base.repo \
http://mirrors.aliyun.com/repo/Centos-7.repo
```

（3）让新的设置生效：

```
[root@localhost ~]# yum makecache
…
Determining fastest mirrors
 * base: mirrors.aliyun.com
 * extras: mirrors.aliyun.com
 * updates: mirrors.aliyun.com
```

如果显示如上，则说明设置成功，继续下一步。

2. 安装 Git

（1）Fabric 使用 Git 作为源码管理工具，可以通过以下命令检查在系统中是否已经存在 Git（如果已经存在 Git，则跳过这一步）：

```
[root@localhost ~]# git --version
```


（2）如果没有安装 Git，则使用 yum 下载安装 Git：

```
[root@localhost ~]# yum -y install git
```

3. 安装 pip

直接使用 yum 安装 pip 一般是不会成功的，这里需要引入一个新的 yum 源——EPEL-Release。EPEL(Extra Packages for Enterprise Linux)是由 Fedora 社区打造的为 RHEL 及衍生发行版如 CentOS、Scientific Linux 等提供高质量软件包的项目。

（1）安装 EPEL 源：

```
[root@localhost ~]# yum -y install epel-release
```

（2）安装 pip：

```
[root@localhost ~]# yum -y install python-pip
```

（3）将 pip 更新到最新版本，yum 源中的 pip 一般不是最新版本，所以在安装后需要更新 pip：

```
[root@localhost ~]# pip install --upgrade pip
```

（4）建议用国内的 pip 源，以加速对 pip 源的访问。

4. 安装 npm

在安装 EPEL 之后，我们也可以通过 yum 下载安装 npm 及 Node.js，直接运行以下命令：

```
[root@localhost ~]# yum -y install npm
```

5. 安装 Docker

（1）查看系统的版本：

```
[root@localhost ~]# uname -r
```

一般来说，64 位系统都支持 Docker 的运行，若想了解 Docker 支持的架构，请参考 Docker 官方文档。

（2）从 yum 源中安装 Docker：

```
[root@localhost ~]# yum -y install docker
```


（3）建议设置 docker hub 镜像来加速下载 Docker 镜像，例如 daocloud 镜像，在注册后可以获得以下加速链接：

```
[root@localhost ~]# curl -sSL https://get.daocloud.io/daotools/ set_mirror.sh | sh
-s http://(yourid).m.daocloud.io
```

如果在通过 daocloud 设置 Docker 加速器后 Docker 服务重启失败，则请编辑 /etc/docker/daemon.json 文件，在将最后一个逗号删除后重启 Docker 服务即可。

（4）在编译的过程中需要得到 docker-compose 功能的支持，因此使用 pip 下载 docker-compose：

```
[root@localhost ~]# pip install docker-compose
```

（5）启动 Docker 服务：

```
[root@localhost ~]# systemctl restart docker.service
```

6. 安装 Golang

Golang 和 pip 一样，在官方 yum 源中并没有提供官方包下载。我们通常可以通过直接下载二进制包进行安装，这里介绍 yum 源的安装方式。通过添加非官方源的方式来管理 Golang 包，可以方便将来的版本升级与相关工具的安装。

（1）首先，增加新 yum 源的 Key：

```
[root@localhost ~]# rpm --import
https://mirror.go-repo.io/centos/RPM-GPG-KEY-GO-REPO
```

（2）然后，将 go-repo.io 的源添加到 yum 源中：

```
[root@localhost ~]# curl -s https://mirror.go-repo.io/centos/go-repo.repo | tee
/etc/yum.repos.d/go-repo.repo
```

（3）使用 yum 安装 Golang：

```
[root@localhost ~]# yum -y install Golang
```

此命令默认会安装最新版本的 Golang，如果希望安装指定版本的 Golang，则运行以下命令：

```
[root@localhost ~]# yum list golang --showduplicates
...
golang.x86_64            1.8.4-0.el7.centos          go-repo
```


```
golang.x86_64            1.8.5-0.el7.centos          go-repo
golang.x86_64            1.8.7-0.el7.centos          go-repo
golang.x86_64            1.9-0.el7.centos            go-repo
golang.x86_64            1.9.1-0.el7.centos          go-repo
golang.x86_64            1.9.2-0.el7.centos          go-repo
golang.x86_64            1.9.3-0.el7.centos          go-repo
golang.x86_64            1.9.4-0.el7.centos          go-repo
golang.x86_64            1.10-0.el7.centos           go-repo
…
```

如果希望使用如上所示的 1.9.4 版本，就执行以下命令：

```
[root@localhost ~]# yum -y install golang-1.9.4-0.el7.centos
```

注意，替换版本号即可。

（4）通过命令检查 Golang 版本：

```
[root@localhost ~]# go version
```

Fabric 要求 Golang 的版本在 1.9 以上。截至 2018 年 8 月，Golang 最新的版本是 1.10，但 Golang 的调试工具 delve 对该版本的支持并不好（参见 GitHub 网站的 derekparker/delve 项目），所以在这里推荐安装 1.9.4 版本。

（5）创建 Golang 项目的源码目录：

```
[root@localhost ~]# mkdir /usr/local/src/go
```

这里，对具体的路径可以根据自己的实际情况进行调整。

（6）编辑 shell 环境变量，添加 GOPATH 和 GOROOT：

```
[root@localhost ~]# echo "export GOPATH=/usr/local/src/go" >> /etc/bashrc
[root@localhost ~]# echo "export GOROOT=/usr/bin/golang" >> /etc/bashrc
[root@localhost ~]# echo "export PATH=$PATH:$GOROOT/bin" >> /etc/bashrc
```

需要注意的是，GOROOT 应该指向 Golang 的二进制安装路径，可以在安装后通过 which go 命令找到二进制存放路径，根据安装方式或者版本的不同，二进制存放路径可能在不同的路径下，例如：

```
[root@localhost ~]# which go
/user/bin/go
[root@localhost ~]# ll /usr/bin/go
lrwxrwxrwx. 1 root root 20 3月   1 07:43 /usr/bin/go -> /etc/alternatives/go
[root@localhost ~]# ll /etc/alternatives/go
```

```
   lrwxrwxrwx. 1 root root 22 3月   1 07:43 /etc/alternatives/go ->
/usr/lib/golang/bin/go
   [root@localhost ~]# /usr/lib/golang/bin/go
   -rwxr-xr-x. 1 root root 11304359 2月  17 15:11 /usr/lib/golang/bin/go
```

这时可以看到实际的安装目录为/usr/lib/golang，我们在设置 GOROOT 时需要将命令修改如下：

```
[root@localhost ~]# echo "export GOROOT=/usr/lib/golang" >> /etc/bashrc
```

编辑/etc/bashrc 主要是为了得到所有用户的支持，如果只需要当前用户的支持，则请编辑~/.bashrc 文件。

（7）使环境变量生效：

```
[root@localhost ~]# source /etc/bashrc
```

如果编辑的是~/.bashrc 文件，则请执行：

```
[root@localhost ~]# source ~/.bashrc
```

（8）检查环境变量是否生效：

```
[root@localhost ~]# echo $GOPATH
```

在输出结果中应该可以看到在文件中指定的路径。

3.1.2　编译 Fabric

Fabric 的源码主要在 Gerrit 下，在 GitHub 网站的 hyperledger/fabric 项目中是只读镜像，如果仅仅想使用代码，则下载 GitHub 代码镜像即可；如果想贡献项目，则最好从 Gerrit（gerrit.hyperledger.org）下载代码。

1. 从 Gerrit 下载源码

（1）登录 Linux 基金会网站并注册 LFID，如图 3-1 所示。

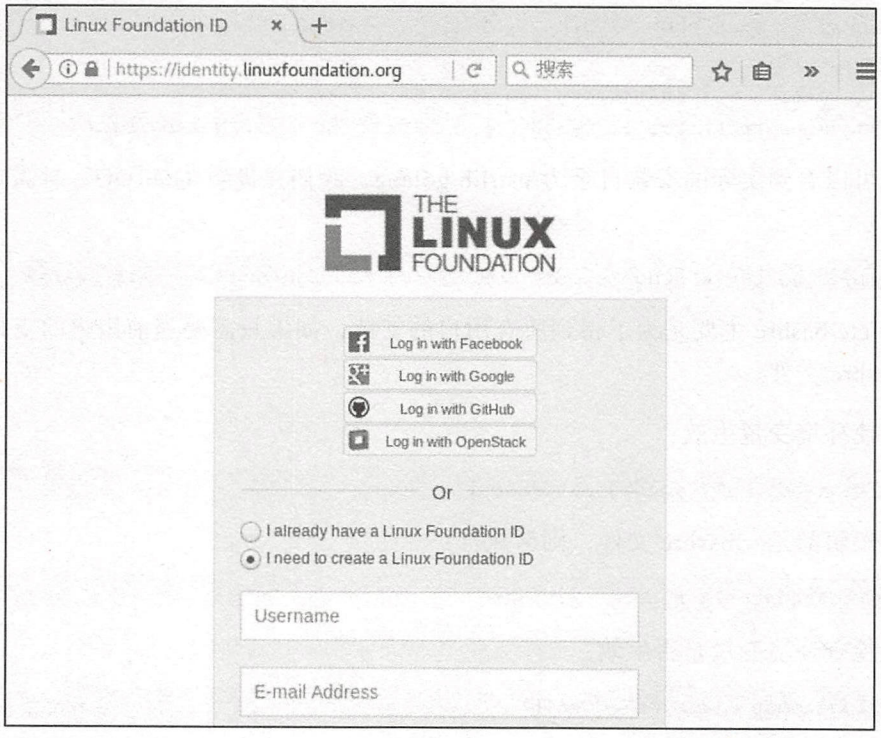


图3-1

在点选“I need to create a Linux Foundation ID”后填写相关信息。

（2）在本机上生成自己的 SSH Key：

```
[root@localhost ~]# ssh-keygen -t rsa -C your@email.com
```

在一般情况下保留默认的生成路径，在提示 Enter passphrase 也就是输入私钥密码时，输入自己常用的密码即可。

（3）复制生成的 SSH Key：

```
[root@localhost ~]# cat ~/.ssh/id_rsa.pub
```

要注意 id_rsa 与 id_rsa.pub 的区别：id_rsa 是私钥，id_rsa.pub 是公钥，这里只能复制公钥，在复制私钥时不会添加成功。

（4）登录 Gerrit，使用刚刚注册的 LFID。

（5）单击右上角的自己的名字，然后选择 Settings，如图 3-2 所示。

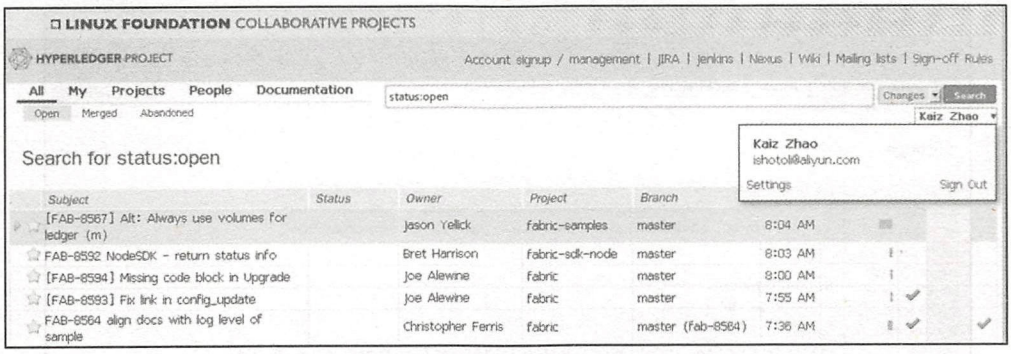


图3-2

在左侧菜单中选择 SSH Public Keys，如图 3-3 所示。

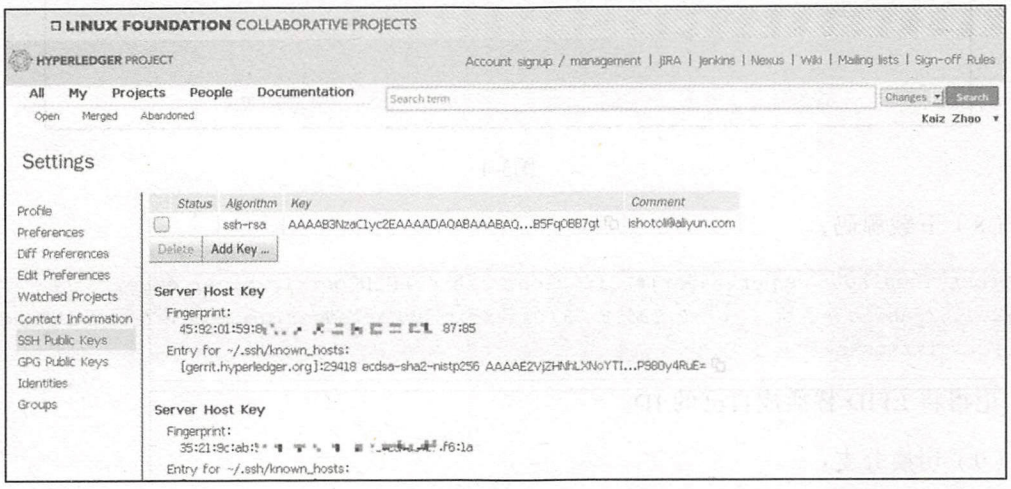


图3-3

然后在 Add SSH Public Key 处粘贴生成的 SSH Key，如图 3-4 所示。

单击 Add 按钮即可生成。

（6）创建源码目录：

```
[root@localhost ~]# mkdir -p $GOPATH/src/github.com/hyperledger
```

（7）进入源码目录：

```
[root@localhost ~]# cd $GOPATH/src/github.com/hyperledger
```

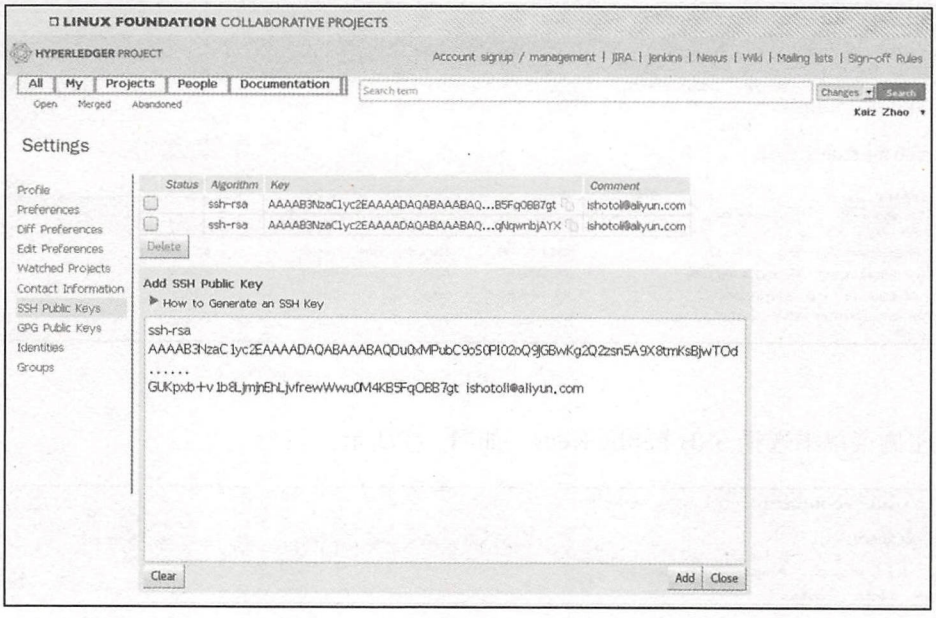


图3-4

（8）下载源码：

```
[root@localhost hyperledger]# git clone ssh://LFID@gerrit.hyperledger.
org:29418/fabric && scp -p -P 29418 LFID@gerrit.hyperledger.org:hooks/ commit-msg
fabric/.git/hooks/
```

记得将 LFID 替换成自己的 ID。

（9）切换分支：

```
[root@localhost hyperledger]# cd fabric
[root@localhost fabric]# git checkout -b remotes/origin/release-1.1
```

因为本书基于 Fabric 1.1 版本进行操作，请务必切换分支到 1.1 版本。

2. 安装工具

在代码下载完成后，下载 BDD 测试依赖工具：

```
[root@localhost fabric]# pip install -r devenv/bddtests-requirements.txt
```

使用 yum 源下载编译依赖项：

```
[root@localhost fabric]#  yum install gcc gcc-c++ make openssl-devel libtool bzip2
libtool-ltdl-devel
```

3. 开始编译

编译步骤如下。

（1）进入源码目录，在开始编译之前请确认工具都已安装完成且 Docker 后台已经启动：

```
[root@localhost fabric]# cd $GOPATH/src/github.com/hyperledger/fabric
```

（2）生成 Golang 编译使用的工具：

```
[root@localhost fabric]# make gotools
```

（3）生成 Docker 版本的 gotools:

```
[root@localhost fabric]# make build/docker/gotools
```

如果生成失败，则需要手动将在 gotools 下编译好的文件复制到以下目录中：

```
[root@localhost fabric]# cp -r gotools/build/gopath/bin/* build/docker/gotools/bin/
```

如果已经生成了 gotools，则可以尝试：

```
[root@localhost fabric]# make linter
```

（4）编译本地二进制文件：

```
[root@localhost fabric]# make native
```

这相当于顺序编译 Peer、Orderer、Configtxgen、Cryptogen、Configtxlator，在 build/bin 目录下可以找到生成的二进制文件。

（5）生成 Docker 镜像：

```
[root@localhost fabric]# make docker
```

至此，编译工作基本完成，生成的 Docker 镜像可以通过 docker images 命令看到。

在编译遇到问题时该怎么办？通常的建议是打开 make 的调试模式，例如：

```
[root@localhost fabric]# make -d peer
```

我们一般可以看到 make 命令当前的操作，debug 的输出量很大，但是能够清楚地显示在失败前进行的操作，便于定位问题。如果出现 mkdir /opt/... : permission denied，则根

本原因是下载的 fabric-baseimage 镜像在运行时没有获取本地挂载目录的写权限。通过以下命令在 docker-env.mk 中搜索：

```
DRUN = docker run -i --rm $(DOCKER_RUN_FLAGS)
```

在这条命令的最后加上--privileged，即可允许在 Docker 内编译时创建本地路径，例如：

```
@@ -49,7 +49,7 @@ endif

 DRUN = docker run -i --rm $(DOCKER_RUN_FLAGS) \
       -v $(abspath .):/opt/gopath/src/$(PKGNAME) \
-      -w /opt/gopath/src/$(PKGNAME)
+      -w /opt/gopath/src/$(PKGNAME) --privileged
```

如果以下命令执行不下去，则在修改 images/javaenv/Dockerfile.in 中对应的地址后尝试重新下载：

```
RUN curl -sSL https://services.gradle.org/distributions/gradle-2.12-bin.zip
```

在遇到以下类似的问题时，有很多人建议用 github.com/golang 替换原来的 golang.org/x 以解决 golang.org/x 无法访问的问题，这样做确实可以通过编译，但从建立 Fabric 开发调试环境的角度来看，将来还需要下载很多 golang.org 下的依赖工具，无法根据需求一一替换路径：

```
package golang.org/x/tools/go/gcexportdata: unrecognized import path
"golang.org/x/tools/go/gcexportdata" (https fetch: Get
https://golang.org/x/tools/go/gcexportdata?go-get=1: dial tcp 216.239.37.1:443: i/o
timeout)
```

因此，还是建议通过设置代理服务器的方式加速下载，例如：

```
export https_proxy=http://your.http.proxy.ip:port/
export all_proxy=socks5://your.socks5.proxy.ip:port/
```

3.1.3 部署 Fabric 网络

在所有代码都编译生成以后，是否想看看 Fabric 的真面目？Fabric 已经预置了一部分示例程序可以让我们直接使用。为了方便在之后开发调试 Orderer、Peer 及 Chaincode，我们也会需要一套运行在本地环境下的 Fabric 网络进行开发。接下来，我们会一步一步地建立只有一个 Orderer 节点和一个 Peer 节点的简单网络。建议在每一步操作之后，在源码中搜索命令或者说明中的一些关键字来阅读源码，以加深对结构的理解。

1. 配置环境

创建/var/hyperledger/目录，并将目录的所有者修改为运行的用户，或者编辑sampleconfig/ orderer.yaml 的 FileLedger/Location 中的路径位置来指向我们期望的路径。建议将$GOPATH/src/github.com/hyperledger/fabric/build/bin 加入环境变量中。

2. 启动 Orderer 节点

首先，我们需要确定当前的命令行在 Fabric 源码目录下，并且 build 目录下的二进制文件都已经生成，如果已经把 fabric/build/bin 加入 PATH 环境变量中，则可以在执行以下命令时省略其中的 build/bin/部分：

```
[root@localhost fabric]# ORDERER_GENERAL_GENESISPROFILE=SampleDevModeSolo
build/bin/orderer
2018-02-07 17:04:42.106 CST [Orderer/main] main -> INFO 001 Starting Orderer:
Version: 1.0.6-snapshot-78e18d17
Go version: go1.9.2
OS/Arch: darwin/amd64
2018-02-07 17:04:42.155 CST [common/configtx/tool/localconfig] Load -> INFO 002
Loaded configuration: /Users/Fanfan/workshop/go/src/github.com/hyperledger/fabric/
sampleconfig/configtx.yaml
2018-02-07 17:04:42.162 CST [fsblkstorage] newBlockfileMgr -> INFO 003 Getting block
information from block storage
2018-02-07 17:04:42.168 CST [Orderer/multichain] NewManagerImpl -> INFO 004 Starting
with system channel testchainid and Orderer type solo
2018-02-07 17:04:42.168 CST [Orderer/main] main -> INFO 005 Beginning to serve requests
```

我们发现，第 1 个启动的节点默认会读取 FABRIC_CFG_PATH 指定的目录，如果不存在 FABRIC_CFG_PATH 的环境变量定义，则会读取 GOPATH 路径下 src/github.com/hyperledger/fabric/sampleconfig 目录的配置文件中的 SampleDevModeSolo 段，对 Orderer 节点进行配置。

另外，通过在源码中搜索 sampleconfig 关键字，我们注意到这些配置文件还出现在 Orderer、Peer 等 Docker 打包文件中，也就是说这套配置文件同时会作为默认的配置文件出现在 Docker 运行环境下。建议在修改这个目录下的文件时格外小心，如果需要调整配置项，则建议通过设置环境变量的方式覆盖配置文件中的变量。

通过查看 orderer/common/localconfig 下的 config.go，我们可以看到主要加载了一个名为 Orderer 的配置文件，所以在 sampleconfig 目录下存在 orderer.yaml 文件，我们会在之后

的章节中深入解释其中的配置项。通过打印的 log，我们也可以看到 Orderer 节点在启动时读取了 configtx.yaml，在 configtx.yaml 中能够找到 SampleDevModeSolo 段的定义。

Orderer 节点在加载完配置之后，会通过 newBlockfileMgr 加载区块文件，并初始化默认的 channel_id（testchainid）与 Orderer 节点的启动类型（Solo）；当网络并没有显式指定通道（Channel）时，所有的交易区块都会被写入 testchainid 中，我们可以在第 1 步配置的目录（例如 /var/hyperledger/production/orderer/chains/）下找到存储区块的文件。没错，在通道名称（例如 testchainid）目录下，正好存放了名为 Blockfile_000000 的文件，这个文件保存的内容就是 Fabric 的创世区块，关于创世区块将会在第 5 章详细解析。

Fabric 的区块链文件是以 Blockfile 为前缀的一系列文件，从 000 000 到 999 999 一共有 100 万个文件，那么每个文件最大为多少呢？在 common/ledger/blkstorage/fsblkstorage /config.go 的 defaultMaxBlockfileSize 及 core/ledger/ledgerconfig/ ledger_config.go 的 GetMaxBlockfileSize 中定义了文件最大为 $64 \times 1024 \times 1024$ 字节，也就是说，同一通道的所有区块文件最大应该是 61TB。如果需要调整大小，则要注意对于文件的大小有两处定义：当需要写入的区块与当前文件的大小之和超过区块文件的最大大小之后，会创建一个新的区块文件，也就是说每个区块文件都不会刚好是 64MB；Solo 模式则意味着不使用 Kafka 服务排序（Orderer 节点的运行当前仅支持 Solo 与 Kafka 两种模式），在启动完成后等待接收消息进行处理。

3. 启动 Peer 节点

新建一个命令行终端，进入 Fabric 源码路径下执行以下命令：

```
   [root@localhost fabric]# build/bin/peer node start --peer-chaincodedev=true
2018-02-07 17:10:58.794 CST [nodeCmd] serve -> INFO 001 Starting peer:
    Version: 1.0.6-snapshot-78e18d17
    Go version: go1.9.2
    OS/Arch: darwin/amd64
    Chaincode:
     Base Image Version: 0.3.2
     Base Docker Namespace: hyperledger
     Base Docker Label: org.hyperledger.fabric
     Docker Namespace: Hyperledger
   2018-02-07 17:10:58.794 CST [ledgermgmt] initialize -> INFO 002 Initializing ledger
mgmt.
   2018-02-07 17:10:58.795 CST [kvledger] NewProvider -> INFO 003 Initializing ledger
provider
   2018-02-07 17:10:58.804 CST [kvledger] NewProvider -> INFO 004 ledger provider
```

```
Initialized
   2018-02-07 17:10:58.804 CST [ledgermgmt] initialize -> INFO 005 ledger mgmt
initialized
   2018-02-07 17:10:58.804 CST [nodeCmd] serve -> INFO 006 Running in Chaincode
development mode
   2018-02-07 17:10:58.804 CST [nodeCmd] serve -> INFO 007 Disable loading validity system
Chaincode
   2018-02-07 17:10:58.808 CST [eventhub_producer] start -> INFO 008 Event processor
started
```

看看发生了什么：第 2 个启动的节点是 Peer 节点，Peer 节点在作为服务器启动时通常同时包含 Endorser 与 Committer 这两个角色。我们在这个环节设置了参数 --peer-chaincodedev=true，这样 Chaincode 的加载与启动就不再由 Peer 节点控制了，而由用户手动控制。通过搜索源码，我们不难看到 Chaincode 的开发模式由 core/Chaincode/Chaincode_ support.go 中的 IsDevMode()控制。

Peer 节点在作为服务启动后进行了必要的初始化，然后等待消息处理。

4. 生成通道配置交易

新建一个终端窗口，然后在 Fabric 源码目录下执行以下命令：

```
   [root@localhost fabric]# build/bin/configtxgen -channelID ch1
-outputCreateChannelTx ch1.tx -profile SampleSingleMSPChannel
   2018-02-21 22:29:26.260 CST [common/configtx/tool] main -> INFO 001 Loading
configuration
   2018-02-21 22:29:26.276 CST [common/configtx/tool] doOutputChannelCreateTx -> INFO
002 Generating new channel configtx
   2018-02-21 22:29:26.276 CST [common/configtx/tool] doOutputChannelCreateTx -> INFO
003 Writing new channel tx
```

这一步主要是使用 Configtxgen 工具生成通道交易配置文件，Configtxgen 会从 configtx.yaml 配置文件中读取 SampleSingleMSPChannel 段，该段的定义如下：

```
   SampleSingleMSPChannel:
         Consortium: SampleConsortium
         Application:
            Organizations:
               - *SampleOrg
```

可以看到，SampleSingleMSPChannel 定义了一个联盟 SampleConsortium，其中只包含

一个组织 SampleOrg；通过 outputCreateChannelTx，我们知道 configtxgen 将配置信息输出到配置交易文件 ch1.tx 中，那么在 ch1.tx 中具体包含哪些内容呢？我们可以通过以下命令查看：

```
[root@localhost fabric]# build/bin/configtxgen -inspectChannelCreateTx ch1.tx
2018-02-22 19:21:14.407 CST [common/configtx/tool] main -> INFO 001 Loading
configuration
2018-02-22 19:21:14.422 CST [common/configtx/tool] doInspectChannelCreateTx -> INFO
002 Inspecting transaction
2018-02-22 19:21:14.428 CST [common/configtx/tool] doInspectChannelCreateTx -> INFO
003 Parsing transaction

Channel creation for channel: ch1

Read Set:
{
   "Channel": {
      "Values": {
         "Consortium": {
            "Version": "0",
            "ModPolicy": "",
            "Value": {
               "name": "SampleConsortium"
            }
         }
      },
      "Policies": {},
      "Groups": {
         "Application": {
            "Values": {},
            "Policies": {},
            "Groups": {
               "SampleOrg": {
                  "Values": {},
                  "Policies": {},
                  "Groups": {}
               }
            }
         }
      }
   }
```

```
}

Write Set:
{
   "Channel": {
      "Values": {
         "Consortium": {
            "Version": "0",
            "ModPolicy": "",
            "Value": {
               "name": "SampleConsortium"
            }
         }
      },
      "Policies": {},
      "Groups": {
         "Application": {
            "Values": {},
            "Policies": {
               "Admins": {
                  "Version": "0",
                  "ModPolicy": "Admins",
                  "Policy": {
                     "PolicyType": "3",
                     "Policy": {
                        "subPolicy": "Admins",
                        "rule": "MAJORITY"
                     }
                  }
               },
               "Writers": {
                  "Version": "0",
                  "ModPolicy": "Admins",
                  "Policy": {
                     "PolicyType": "3",
                     "Policy": {
                        "subPolicy": "Writers",
                        "rule": "ANY"
                     }
                  }
               },
```

```
                "Readers": {
                    "Version": "0",
                    "ModPolicy": "Admins",
                    "Policy": {
                        "PolicyType": "3",
                        "Policy": {
                            "subPolicy": "Readers",
                            "rule": "ANY"
                        }
                    }
                }
            },
            "Groups": {
                "SampleOrg": {
                    "Values": {},
                    "Policies": {},
                    "Groups": {}
                }
            }
        }
    }
}

Delta Set:
[Groups] /Channel/Application
[Policy] /Channel/Application/Admins
[Policy] /Channel/Application/Writers
[Policy] /Channel/Application/Readers
```

可以看到，在 ch1.tx 中主要是策略定义，包含 Read Set、Write Set 两组策略。在 Read Set 中并没有明显的策略限制，而在 Write Set 中定义了以下三条策略：

- Channel.Groups.Application.Policies.Admins；
- Channel.Groups.Application.Policies.Writes；
- Channel.Groups.Application.Policies.Readers。

这三条策略被显示在 Delta Set 中，被记为类似于/Channel/Application/Admins 的形式，其中/Channel/Application 是组名，Admins 是策略名。在这里写入的策略（主要用于定义规则的读写权限规则）都是 Configtxgen 自带的默认策略。

在使用 Fabric 时可以不定义通道吗？答案是可以。Orderer 节点在启动时会生成默认的通道 testchainid，Peer 客户端在不使用-c 指定通道时也会选择默认的通道，但从设计上来说并不建议使用默认的通道，之后会讲解具体的原因。

5. 创建通道

在当前窗口继续执行以下指令：

```
[root@localhost fabric]# build/bin/peer channel create -o 127.0.0.1:7050 -c ch1 -f ch1.tx
2018-02-07 17:30:16.617 CST [channelCmd] InitCmdFactory -> INFO 001 Endorser and Orderer connections initialized
2018-02-07 17:30:16.659 CST [channelCmd] InitCmdFactory -> INFO 002 Endorser and Orderer connections initialized
2018-02-07 17:30:16.862 CST [main] main -> INFO 003 Exiting...
```

这时在 Orderer 终端可以看到：

```
2018-02-21 22:29:33.450 CST [fsblkstorage] newBlockfileMgr -> INFO 007 Getting block information from block storage
2018-02-21 22:29:33.451 CST [Orderer/multichain] newChain -> INFO 008 Created and starting new chain ch1
```

我们将创建通道的命令提交给 Orderer 节点，并提交了通道配置交易文件 ch1.tx，Orderer 节点在这个新的通道中建立了新的 BlockfileMgr，生成了创世区块并新建了对应的 chainSupport，之后会继续等待消息；Peer 客户端则在 Orderer 节点成功创建通道后下载创世区块。创世区块是从哪里来的呢？Orderer 在创建通道之后，会把配置消息的报文作为有效载荷来创建创世区块；Peer 客户端在下载创世区块后，会将其以{channelID}.block 为文件名的格式保存到本地，在这里 Peer 客户端输出的创世区块文件为 ch1.block。同样，创建通道使用的配置内容虽然被打包为交易，但并没有与普通交易一样经过 Peer 节点的背书，而是通过 Gossip 广播直接提交给 Orderer 节点处理，这个流程和其他处理流程不太一样。我们可以通过以下命令查看 ch1.block 的内容：

```
[root@localhost fabric]# build/bin/configtxgen -inspectBlock ch1.block
```

由于输出的内容较多，所以这里只对总体结构进行大概描述，导出的 JSON 结构如图 3-5 所示。

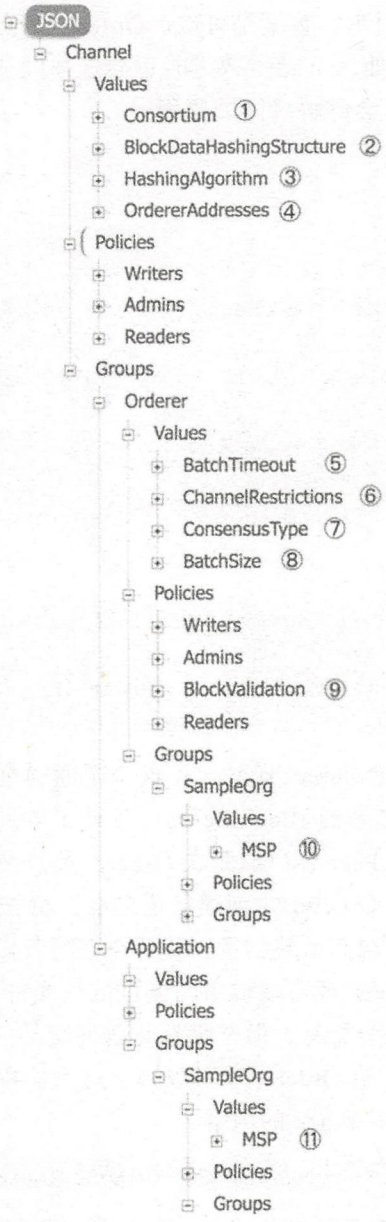


图3-5

通道的配置区块文件定义了通道的配置 Values、通道策略 Policies 与通道组织 Groups，除了策略定义，配置还主要包含以下内容：

（1）联盟的名称；

（2）在使用 Merkle Tree 计算 BlockData 哈希值时需要的宽度；

（3）哈希算法，一般是 SHA256；

（4）Orderer 节点的地址；

（5）批处理的超时时长；

（6）创建通道的数量限制，0 指没有限制；

（7）共识类型，指是 Solo 还是 Kafka；

（8）批处理的大小，包括交易数量与区块的大小；

（9）区块验证策略；

（10）与（11）分别定义了节点的 MSP，主要包含节点对应的证书。

以上内容构成了整个通道的基本配置。

6. 加入通道

接下来，我们将 Peer 节点加入通道中：

```
[root@localhost fabric]# build/bin/peer channel join -b ch1.block
2018-02-07 18:24:14.001 CST [channelCmd] InitCmdFactory -> INFO 001 Endorser and
Orderer connections initialized
2018-02-07 18:24:14.024 CST [channelCmd] executeJoin -> INFO 002 Peer joined the
channel!
2018-02-07 18:24:14.024 CST [main] main -> INFO 003 Exiting...
```

这时可以在 Peer 终端看到：

```
2018-02-21 22:33:02.575 CST [ledgermgmt] CreateLedger -> INFO 023 Creating ledger
[ch1] with genesis block
2018-02-21 22:33:02.575 CST [fsblkstorage] newBlockfileMgr -> INFO 024 Getting block
information from block storage
2018-02-21 22:33:02.578 CST [kvledger] Commit -> INFO 025 Channel [ch1]: Created block
[0] with 1 transaction(s)
2018-02-21 22:33:02.579 CST [ledgermgmt] CreateLedger -> INFO 026 Created ledger [ch1]
with genesis block
2018-02-21 22:33:02.585 CST [cscc] Init -> INFO 027 Init CSCC
```

```
    2018-02-21 22:33:02.585 CST [sccapi] deploySysCC -> INFO 028 system Chaincode
cscc/ch1(github.com/hyperledger/fabric/core/scc/cscc) deployed
    2018-02-21 22:33:02.585 CST [sccapi] deploySysCC -> INFO 029 system Chaincode
lscc/ch1(github.com/hyperledger/fabric/core/scc/lscc) deployed
    2018-02-21 22:33:02.585 CST [escc] Init -> INFO 02a Successfully initialized ESCC
    2018-02-21 22:33:02.585 CST [sccapi] deploySysCC -> INFO 02b system Chaincode
escc/ch1(github.com/hyperledger/fabric/core/scc/escc) deployed
    2018-02-21 22:33:02.585 CST [sccapi] deploySysCC -> INFO 02c system Chaincode
vscc/ch1(github.com/hyperledger/fabric/core/scc/vscc) deployed
    2018-02-21 22:33:02.585 CST [qscc] Init -> INFO 02d Init QSCC
    2018-02-21 22:33:02.585 CST [sccapi] deploySysCC -> INFO 02e system Chaincode
qscc/ch1(github.com/hyperledger/fabric/core/Chaincode/qscc) deployed
    2018-02-21 22:33:02.586 CST [eventhub_producer] SendProducerBlockEvent -> INFO 02f
Channel [ch1]: Sending event for block number [0]
```

peer channel join 命令会将指定的 Peer 节点加入指定的通道中，Peer 节点通常是通过环境变量 CORE_PEER_ADDRESS 或者 core.yaml 文件中的 peer.address 来指定的，这里并没有设置环境变量，所以保留了配置文件中的设置。虽然在 Orderer 节点中保存着创世区块，但当前架构并没有创世区块的同步流程，因此 Peer 节点在加入通道时需要自带创世区块文件 ch1.block。细心一些可能会发现，和 peer channel create 不同，peer channel join 并没有指定 Orderer 节点。与上一步不同的是，在这一步中 Peer 客户端不再和 Orderer 节点有直接的交互，Peer 节点在根据配置文件中的 Orderer 地址建立链接，并且 Endorser 经过 ESCC 背书验证后，将加入通道的请求提交给 Orderer 节点。在 Orderer 节点通过策略验证之后，Peer 节点会初始化对应通道的 CSCC、LSCC、ESCC、VSCC 与 QSCC，之后通道的初始化就基本完成了。

如果在加入的过程中发生以下错误：

```
Error: proposal failed (err: rpc error: code = Unavailable desc = transport is closing)
```

则可能是因为没有定位到正确的区块文件，或者没有连接到正确的 IP 端口地址；可以考虑将-b 参数指向的文件路径改为有效的完整文件路径，并将 sampleconfig/core.yaml 中 Peer 段 address 的地址由 0.0.0.0:7051 改为 127.0.0.1:7051，或者使用以下命令启动：

```
    [root@localhost fabric]# CORE_PEER_ADDRESS=127.0.0.1:7051  build/bin/peer channel
join -b /full/path/to/ch1.block
```

至此，我们的 Fabric 网络部署全部结束。以上是 Fabric 的基础内容，更复杂的用法主要在整体架构层面，在流程方面并不会有太多变化，熟悉这个流程对于之后搭建更复杂的结构与深入代码都会有很大的帮助。

3.2　Fabric 开发调试

在本节中，我们将在之前搭建好的 Fabric 网络上体验智能合约的部署和调用流程，然后给出在开发环境下进行 Fabric 源码调试所需的开发环境搭建与使用指南。

3.2.1　智能合约体验

1. 安装 Chaincode

peer chaincode install 命令主要用于将指定的 Chaincode 代码打包成 ChaincodeDeploymentSpec 格式，进行签名并发送给指定的 Peer 节点。Peer 节点在进行解包并验证签名后，会将有效载荷（Chaincode 打包代码）交给 LSCC 校验并保存，在 LSCC 返回成功后安装流程就结束了。在这一步指定了 Chaincode 的名称为 mycc 及版本为 0：

```
   [root@localhost fabric]# build/bin/peer chaincode install -n mycc -v 0 -p
github.com/hyperledger/fabric/examples/chaincode/go/chaincode_example02
   2018-02-08 07:40:45.459 CST [ChaincodeCmd] checkChaincodeCmdParams -> INFO 001 Using
default escc
   2018-02-08 07:40:45.459 CST [ChaincodeCmd] checkChaincodeCmdParams -> INFO 002 Using
default vscc
   2018-02-08 07:40:46.856 CST [main] main -> INFO 003 Exiting...
```

2. 启动 Chaincode

进入 Chaincode 代码所在的目录，启动 Chaincode：

```
   [root@localhost fabric]# cd examples/chaincode/go/chaincode_example02
   [root@localhost chaincode_example02]# go build chaincode_example02.go
   [root@localhost chaincode_example02]# CORE_CHAINCODE_ID_NAME=mycc:0
CORE_PEER_ADDRESS=127.0.0.1:7052
CORE_CHAINCODE_LOGGING_LEVEL=DEBUG ./chaincode_example02
   2018-02-08 07:42:05.848 CST [shim] SetupChaincodeLogging -> INFO 001 chaincode log
level not provided; defaulting to: INFO
   2018-02-08 07:42:05.848 CST [shim] SetupChaincodeLogging -> INFO 002 chaincode (build
level: ) starting up ...
```

在一般情况下，这个步骤应该是在初始化 Chaincode 时自动完成的，但因为我们是以 peer-chaincodedev 模式启动的，所以需要手动启动 Chaincode。需要注意的是，要确保先

启动 Chaincode，再初始化 Chaincode，否则无法将初始化时的参数传递给 Chaincode。注意，这里连接的地址是 Fabric 1.1 的 Peer 节点的地址，如果签出（Checkout）的 Fabric 源码版本是 1.0.X，则需要将 CORE_PEER_ADDRESS 改为 127.0.0.1:7051。

在这里我们看到了一个与 Chaincode 密不可分的对象：Shim。Shim 指“垫片”，这也很符合它的定位：将 Peer 节点与 Chaincode 隔离开来，并在两者之间相互调用、传递消息。Chaincode 都是运行在 Shim 之内的，Shim 通过 gRPC 与 Peer 节点通信，通过有限状态机（Finite State Machine，FSM）维护 Chaincode 的运行状态。

3. 初始化 Chaincode

新建一个终端，然后在 Fabric 目录下运行以下命令：

```
[root@localhost fabric]# build/bin/peer chaincode instantiate -n mycc -v 0  -c
'{"Args":["init","a","100","b","200"]}' -o 127.0.0.1:7050 -C ch1
2018-02-08 07:42:45.697 CST [ChaincodeCmd] checkChaincodeCmdParams -> INFO 001 Using
default escc
2018-02-08 07:42:45.697 CST [ChaincodeCmd] checkChaincodeCmdParams -> INFO 002 Using
default vscc
2018-02-08 07:42:45.709 CST [main] main -> INFO 003 Exiting...
```

在 Peer 终端可以看到：

```
2018-02-21 22:37:24.539 CST [kvledger] Commit -> INFO 030 Channel [ch1]: Created block
[1] with 1 transaction(s)
2018-02-21 22:37:24.540 CST [eventhub_producer] SendProducerBlockEvent -> INFO 031
Channel [ch1]: Sending event for block number [1]
```

在 Chaincode 终端可以看到：

```
ex02 Init
Aval = 100, Bval = 200
```

在这一步，我们通过 peer chaincode instantiate 对 ch1 通道下名为 mycc 的版本为 0 的 Chaincode 进行了初始化，初始化的消息在被发送给 Peer 节点之后经过 LSCC 验证，对应的 Chaincode 程序就启动了。但处理并没有到此为止，我们看到通过-c 还传递了初始化参数。初始化参数就是设置两个账户 a 与 b，然后分别为两个账户赋值 100 与 200。初始化状态的消息紧接着被提交给刚刚创建的 Chaincode，ESCC 在背书运行结果后通知 Peer 客户端；Peer 客户端再将消息广播给 Orderer 节点；Orderer 节点在进行必要的策略验证后生成区块，然后将区块广播给 Committer；Committer 在通过 VSCC 验证区块后将其存入区块

文件并更新状态数据库。这样我们就有了除创世区块外的第 1 个区块，用于初始化状态。

可以看到，系统中的各个角色都出现了：Orderer、Peer、Endorser、Committer、Chaincode、Shim、LSCC、ESCC 和 VSCC 等。接下来进行真正的业务处理——开始交易。

4. 开始交易

执行以下命令：

```
[root@localhost fabric]# build/bin/peer chaincode invoke -n mycc -c -v0
'{"Args":["invoke","a","b","10"]}' -o 127.0.0.1:7050 -C ch1
2018-02-08 07:43:24.567 CST [ChaincodeCmd] checkChaincodeCmdParams -> INFO 001 Using
default escc
2018-02-08 07:43:24.567 CST [ChaincodeCmd] checkChaincodeCmdParams -> INFO 002 Using
default vscc
2018-02-08 07:43:24.573 CST [ChaincodeCmd] ChaincodeInvokeOrQuery -> INFO 003
Chaincode invoke successful. result: status:200
2018-02-08 07:43:24.573 CST [main] main -> INFO 004 Exiting...
```

在 Peer 终端可以看到：

```
2018-02-21 22:39:29.090 CST [kvledger] Commit -> INFO 032 Channel [ch1]: Created block
[2] with 1 transaction(s)
2018-02-21 22:39:29.091 CST [eventhub_producer] SendProducerBlockEvent -> INFO 033
Channel [ch1]: Sending event for block number [2]
```

在 Chaincode 终端可以看到：

```
ex02 Invoke
Aval = 90, Bval = 210
```

执行 peer chaincode invoke 命令，会进入以下最常用的流程中。

（1）在 Peer 客户端将-c 所带的参数打包签名后发送给 Endorser。

（2）Endorser 在验证签名的有效性之后通过 LSCC 获得对应的 Chaincode。

（3）Chaincode 将参数带入 Invoke 指令执行，返回执行结果。

（4）Endorser 将执行结果提交给 ESCC 进行验证。

（5）ESCC 在验证结束后将结果返回给 Peer 客户端。

（6）Peer 客户端将 ProposalResult 打包成交易并通过原子广播发送给 Orderer 节点，

在发送完成后，Peer 客户端运行结束。

（7）Orderer 节点在验证策略后，等待 Batch 条件达成，在这里是在时间（2s）结束后生成区块，区块的 blocknumber 为 2，是系统的第 2 个区块。

（8）Orderer 节点通过原子广播将新区块广播给 Committer。

（9）Committer 在收到广播后进行签名验证，在验证结束后将区块提交给 VSCC 验证。

（10）VSCC 验证通过在区块中打包的全部交易后，返回成功。

（11）Committer 将交易写入区块文件中并更新状态数据库。

（12）交易结束。

在当前执行的命令中主要是将 a 向 b 转移 10，Chaincode 模拟交易的结果是 A 为 90、B 为 210，我们来看看结果是否正确。

5. 查看交易

执行以下命令：

```
   [root@localhost fabric]# build/bin/peer chaincode query -n mycc -v0 -c
'{"Args":["query","a"]}' -o 127.0.0.1:7050 -C ch1
   2018-02-08 07:44:37.222 CST [ChaincodeCmd] checkChaincodeCmdParams -> INFO 001 Using
default escc
   2018-02-08 07:44:37.222 CST [ChaincodeCmd] checkChaincodeCmdParams -> INFO 002 Using
default vscc
Query Result: 90
   2018-02-08 07:44:37.226 CST [main] main -> INFO 003 Exiting...
```

peer chaincode query 虽然只是查询，并不产生交易，但实际上其提交流程与 invoke 相同，invoke 的第 1 ~ 6 步与 query 相同，但 query 在收到 ProposalResult 之后不会再进行广播，而是在打印 Query Result 之后结束。

到这里，Fabric 的初步体验就结束了，大家是否意犹未尽呢？虽然这是一个小 Demo，但依然有不少发挥空间，例如：

（1）尝试使用默认通道（testchainid），也就是 Peer 客户端不指定通道的名称；

（2）修改日志级别，输出更多的信息；

（3）将每一步产生的结果（配置交易文件、区块文件、LevelDB 等）导出，查看其中

的内容；

（4）尝试修改配置文件，观察其对流程与日志的影响；

（5）尝试加载不同的 Chaincode，产生新的交易方式；

（6）尝试添加更多的通道；

（7）尝试添加 Kafka 及 CouchDB。

3.2.2　调试 Fabric 源码

之前讲解了 Fabric 的基本流程，接下来搭建一个 Fabric 调试环境，便于我们更好地学习与开发。

Golang 相对于传统的 Java、C++来说还是一门新语言，在 IDE 方面也缺乏 VisualStudio、Eclipse、IDEA 等大厂的原生支持。得益于 Golang 活跃的开源社区，我们依然可以获得好用的调试工具，其中比较出色的是微软的 VSCode，如图 3-6 所示。

图3-6

作为一个由开源社区支持的轻量级跨平台编辑器，VSCode 支持 Mac、Linux、Windows 等常见的桌面操作系统，开放的插件机制对 Golang、Python、Node.js 等各种编程语言及

框架也都有良好的支持，通过 VSCode 内置的调试框架能很方便地开发出对应语言的好用的调试工具。从启动速度、内存及 CPU 占用来看，VSCode 也比同类型软件更有优势。因此，这里以 VSCode 为例，利用开源社区提供的工具链搭建一个 Fabric 调试环境。

1. 下载安装 VSCode

VSCode 可以从 Visual Studio Code 网站下载，下载界面如图 3-7 所示。

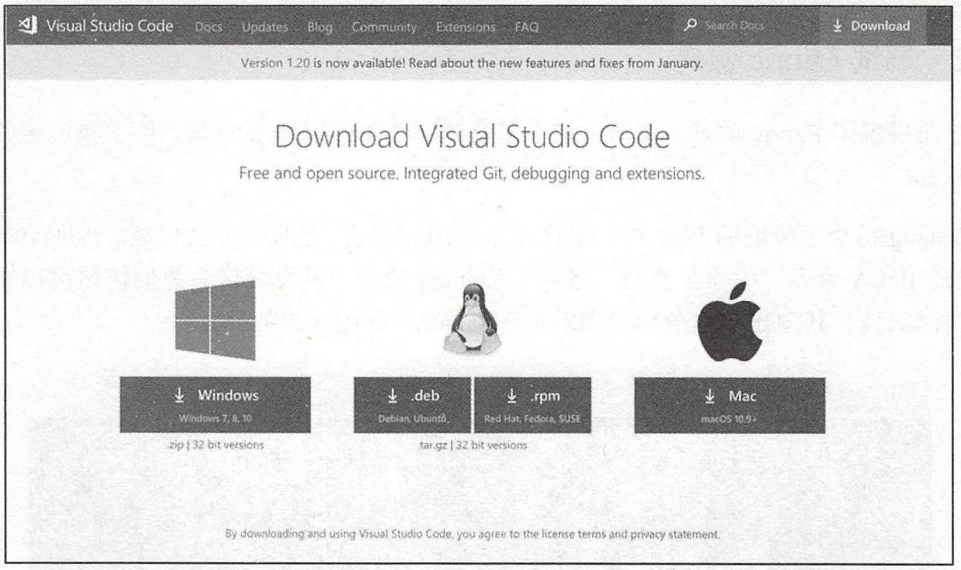


图3-7

我们通过网页下载对应的安装包并安装即可，需要注意的是，建议将 VSCode 配合 Git 2.0 以上版本运行，而部分 Linux 版本（例如 CentOS）内置的 Git 版本为 1.8，需要在更新 Git 版本之后才能配合 VSCode 运行。

2. 下载 Golang 插件

在 VSCode 界面通过 Ctrl+Shift+X 组合键打开扩展界面，搜索“go”，得到的第 1 个搜索结果就是 Golang 语言的支持插件，单击“安装”按钮后重新加载 VSCode 以使插件生效，如图 3-8 所示。

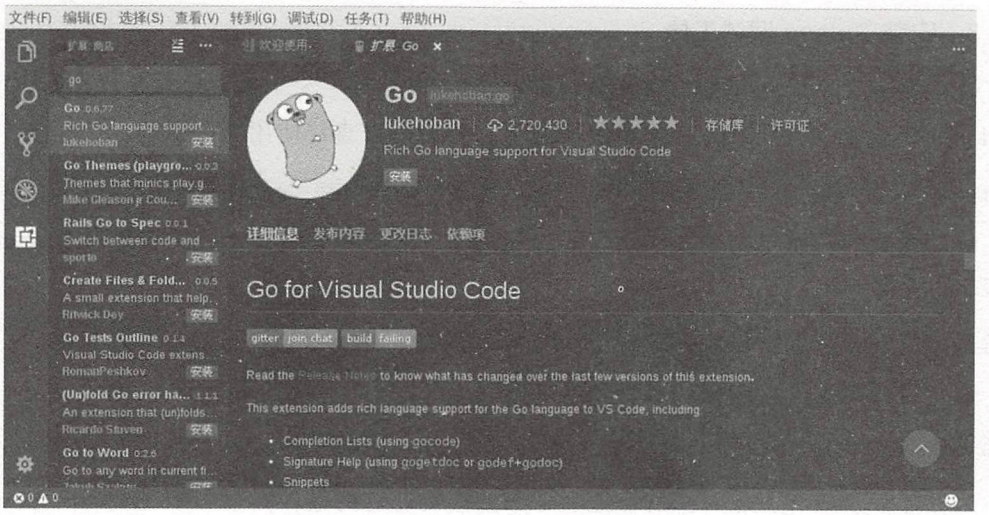


图3-8

Go for VSCode 插件的主要功能有：语法高亮、代码格式化、代码补全、自动导入、跳转定义、查找引用/实现、自动生成单元测试框架、语义/语法错误提示，等等。整合的工具链已能满足大多数开发需求，这些配置可以通过 Ctrl+,组合键（在 Mac 下是 Command+,组合键）打开用户设置界面，对各种功能进行配置，如图 3-9 所示。

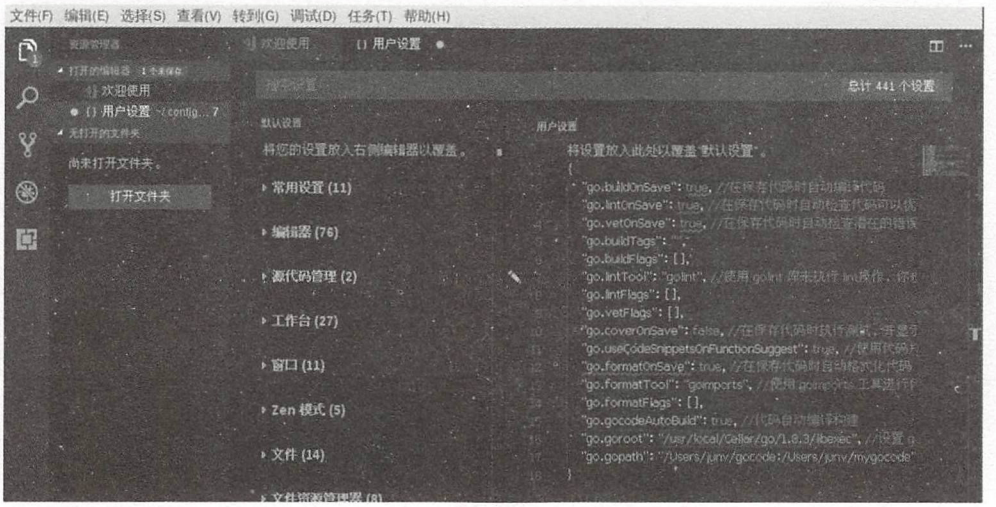


图3-9

3. 安装 Golang 调试工具

VSCode 的 Golang 的调试功能同样依赖于一个开源的第三方工具 delve(在 GitHub 网站的 derekparker/delve 项目中)，在大多数情况下，我们可以直接使用命令安装：

```
[root@localhost ~]# go get -u github.com/derekparker/delve/cmd/dlv
```

Mac 系统因为系统安全的限制，通过 go get 方式不能安装证书，推荐通过 HomeBrew 安装：

```
$ brew install go-delve/delve/delve
```

或者将源码克隆（Clone）到本地后执行：

```
$ make install
```

这样才能生成和安装对应的自签名证书。需要注意的是，截至 2018 年 3 月，delve 尚不支持对 Golang 1.10 版本的调试，所以推荐安装 Golang 1.9.X 版本。

4. 安装插件

在上述软件都安装成功后，就可以通过 Ctrl+O 组合键选择 Fabric 源码目录了，如图 3-10 所示。

图3-10

在单击“确定”按钮后打开文件夹，然后选择其中的一个文件，这时插件会提示安装对应的依赖工具，选择 Install All，如图 3-11 所示。

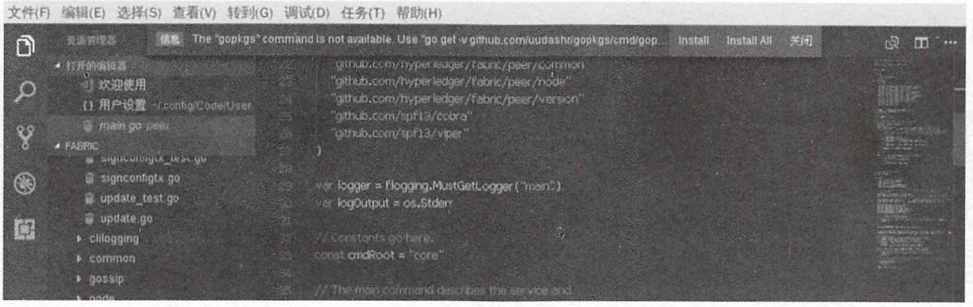


图3-11

需要注意的是，这些工具大多通过 go get 下载，如果之前在编译 Fabric 时将 golang.org/x 目录指向了 github.com/golang，对工具的下载就会失败，所以还是建议通过代理服务器的形式下载依赖插件，如果安装失败且需要手动重新安装这些插件，则需要通过 go get 安装这些安装包，例如 gocode、gopkgs、go-outline、go-symbols、guru、gorename、godef、godoc、goreturns、golint 和 dlv。

5. 设置调试参数

在工具都安装成功后，我们就可以配置调试参数了。首先通过 Ctrl+Shift+D 组合键进入调试界面，如图 3-12 所示。

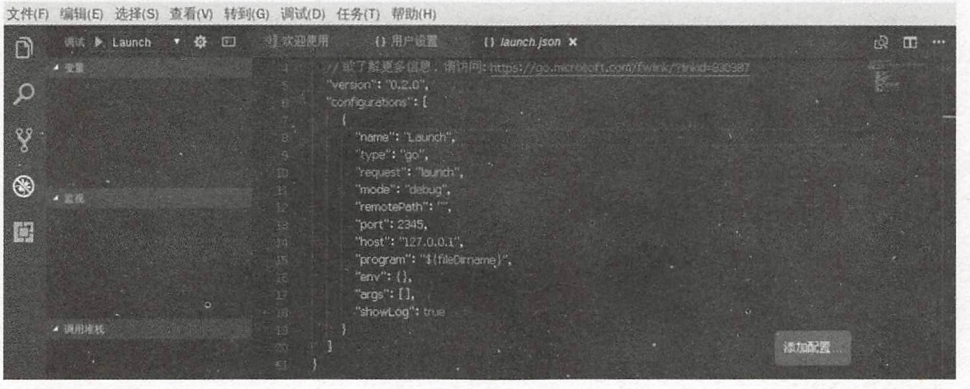


图3-12

单击调试栏的小齿轮图标，即可打开项目的 launch.json 配置文件。在这里可以配置多个项目进行调试配置，因为在 Fabric 中存在多个角色，例如 Peer 客户端、Peer 节点、Orderer 节点、Chaincode 等，因此我们可以配置多个调试项目。这里列举几个常见的配置并进行

简单说明：

```
{
    "name": "peer node start",                    ①
    "type": "go",
    "request": "launch",
    "mode": "debug",                              ②
    "program": "${workspaceRoot}/peer",           ③
    "port": 2345,
    "host": "127.0.0.1",
    "env": {},
    "args": ["node","start","--peer-chaincodedev=true","--logging-level=DEBUG"],④
    "showLog": true,
    "backend": "native"                           ⑤
},
{
    "name": "peer Chaincode invoke",
    "type": "go",
    "request": "launch",
    "mode": "debug",
    "program": "${workspaceRoot}/peer",
    "port": 2345,
    "host": "127.0.0.1",
    "env": {},
    "args": ["Chaincode", "invoke", "-n=mycc","-c='{\"Args\":[\"invoke\",\"a\",\"b\
",\"10\"]}","-o=127.0.0.1:7050","-C=ch1", "--logging-level=DEBUG"],
    "showLog": true,
    "backend": "native"
},
{
    "name": "Orderer",
    "type": "go",
    "request": "launch",
    "mode": "debug",
    "program": "${workspaceRoot}/orderer",
    "port": 2345,
    "host": "127.0.0.1",
    "env": {"ORDERER_GENERAL_GENESISPROFILE":"SampleDevModeSolo"}, ⑥
    "args": [],
    "showLog": true,
    "backend": "native"
```

```
}
```

其中，

①指调试配置的名称，使用易于理解的名称即可；

②指调试模式，debug 代表编译 program 下的内容，然后使用调试器运行；test 用来调试测试程序；exec 代表运行已经编译好的指定文件；remote 则是连接（attach）到指定的远程调试服务上，远程服务的地址和端口由 remotePath、host 与 port 参数指定；

③指调试程序，一般来说是 main.go 程序所在的目录，这里是当前工作根目录（workspaceRoot）下的 Peer 目录，也可以指定具体要执行的文件，默认配置下的${file}指代当前文件；

④指运行参数，在参数名称和值之间建议用等号连接；

⑤指 delve 调试参数，处理部分版本的 delve 配合 VSCode 进行本地调试时报错的问题；

⑥指环境变量，注意与运行参数的定义差异，运行参数是字符串数组，而环境变量是包含若干键值对的对象。

在编辑完 launch.json 后，我们就可以开始调试了，在选择对应的调试配置后按 F5 键即可启动调试，如图 3-13 所示。

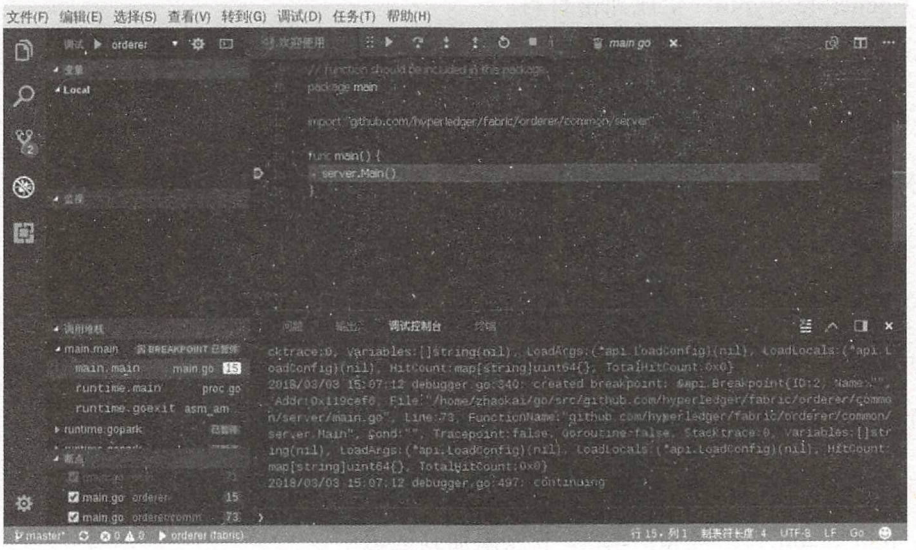


图3-13

另外，F9 键用于设置或移除断点，F10 键用于单步跳过，F11 键用于单步执行，Shift+F11 组合键用于单步跳出，F5 键用于继续执行，若经常使用 Visual Studio，则会感觉格外亲切！在 VSCode 自带的调试框架中除了提供了断点与调试的功能，还提供了查看当前变量、监视变量、调用堆栈的功能。

3.3 更复杂的 Fabric 网络

在对 Fabric 网络有了初步了解之后，我们可以尝试建立一个更复杂的 Fabric 网络。打开本书第 1 章提到的 Fabric_Demo 虚拟机，在/hyperledger/scripts 下有一个 2orgs_network.sh 脚本文件，通过这个脚本文件，我们可以建立一个包含两个组织、四个节点的 Fabric 网络。与第 1 章讲到的两个小型 Fabric 网络不同，这个网络的节点数更多，结构更复杂，而且我们会对参数和配置进行更详细的讲解，方便了解与网络设置相关的知识。

同第 2 章的操作一样，在/hyperledger/scripts 目录下调用 init.sh 初始化 Docker 之后执行 2orgs_network.sh，即可看到它经历了以下步骤。

（a）clear states：清理运行状态（删除 shared 与 statedata 目录下的内容）。

（b）clear containers：清理容器。

（c）generate crypto：生成加密材料。

（d）generate genesis：生成创世区块。

（e）start orderer：启动 Orderer 节点。

（f）start peer1：启动 org1peer1 节点。

（g）start peer2：启动 org1peer2 节点。

（h）start peer3：启动 org2peer1 节点。

（i）start peer4：启动 org2peer2 节点。

（j）create channel：创建通道。

（k）join channel1：将 org1peer1 节点添加到通道中。

（l）join channel2：将 org1peer2 节点添加到通道中。

（m）join channel3：将 org2peer1 节点添加到通道中。

（n）join channel4：将 org2peer2 节点添加到通道中。

（o）updateAnchor1：将 org1peer1 设置为 anchor。

（p）updateAnchor2：将 org2peer1 设置为 anchor。

（q）install chaincode：安装 Chaincode 到通道中。

（r）input commands：启动 docker bash。

下面对这些步骤进行详细讲解。

3.3.1　网络的结构与定义

与第 2 章的双节点例子相比，这里多出来的主要步骤是其他三个 Peer 节点的启动、加入通道操作及两个组织的 updateAnchor 操作（见步骤 f、g、h、i、o、p）。首先，我们看看这个例子的网络拓扑结构，如图 3-14 所示。

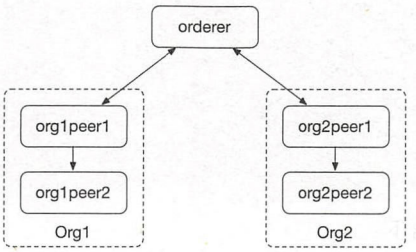


图3-14

在步骤 c、d 中通过配置文件定义了这个网络结构，读取的配置文件分别是 2orgs-config.yaml 与 2orgs-configtx.yaml，2orgs-config.yaml 的内容结构如下：

```
OrdererOrgs:                                    ①
  - Name: Orderer
    Domain: example.com
    CA:
       Country: CN
       Province: Beijing
       Locality: Beijing
    Specs:
```

```
    - Hostname: orderer

PeerOrgs:                                          ②
  - Name: Org1                                     ③
    Domain: org1.example.com
    EnableNodeOUs: true
    CA:
        Country: CN
        Province: Hubei
        Locality: Wuhan
    Template:
      Count: 2
    Users:
      Count: 1

  - Name: Org2                                     ④
    Domain: org2.example.com
    EnableNodeOUs: true
    CA:
        Country: CN
        Province: Beijing
        Locality: Beijing
    Template:
      Count: 2
    Users:
      Count: 1
```

对该文件内容的解释如下。

①定义了 Orderer 组织的信息，包括组织的名称（Name）、所在的域（Domain）、证书信息（CA）及主机名（Hostname）等。

②定义了 Peer 组织的信息，我们可以看到主要包含了 Org1 与 Org2 这两个组织。

③定义了 Org1 组织的信息，包括组织名称（Name）、域名（Domain）、证书信息（CA）、生成的节点个数（Template）、生成的用户数量（Users）及是否生成节点组织信息（EnableNodeOUs）。EnableNodeOUs 似乎比较让人费解，其中的 OU 是 Organization Unit 的缩写，这个开关主要决定在生成 MSP 加密材料时是否包含节点信息，并不影响网络的运行，在这个开关变化时我们可以留意在 crypto-config 目录下生成的 config.yaml 文件及 CA 证书内部的信息。

④定义了 Org2 组织的信息，结构和 Org1 类似，名称和域不同。

在步骤 c 中，cryptogen 工具通过 2orgs-config.yaml 的配置信息生成了 crypto-config 目录，其中包含 5 个节点的 MSP 加密材料及两个用户信息，这些都在接下来的 2orgs-configtx.yaml 配置中用到了。2orgs-configtx.yaml 文件的内容如下：

```
Profiles:
    TwoOrgsOrdererGenesis:                    ①
        Capabilities:
            <<: *ChannelCapabilities
        Orderer:
            <<: *OrdererDefaults
            Organizations:
                - *OrdererOrg
            Capabilities:
                <<: *OrdererCapabilities
        Consortiums:
            SampleConsortium:
                Organizations:
                    - *Org1
                    - *Org2


    TwoOrgsChannel:                           ②
        Consortium: SampleConsortium
        Application:
            <<: *ApplicationDefaults
            Organizations:
                - *Org1
                - *Org2
            Capabilities:
                <<: *ApplicationCapabilities
Organizations:
    - &OrdererOrg                             ③
        Name: OrdererOrg
        ID: OrdererMSP
        MSPDir: crypto-config/ordererOrganizations/example.com/msp

    - &Org1                                   ④
        Name: Org1
        ID: Org1MSP
```

```
        MSPDir: crypto-config/peerOrganizations/org1.example.com/msp
        AnchorPeers:
            - Host: YOUR_INTERNAL_IP
              Port: 5010

    - &Org2
        Name: Org2
        ID: Org2MSP
        MSPDir: crypto-config/peerOrganizations/org2.example.com/msp
        AnchorPeers:
            - Host: YOUR_INTERNAL_IP
              Port: 5014

Orderer: &OrdererDefaults                          ⑤
    OrdererType: solo
    Addresses:
        - YOUR_INTERNAL_IP:31010
    BatchTimeout: 2s
    BatchSize:
        MaxMessageCount: 10
        AbsoluteMaxBytes: 98 MB
        PreferredMaxBytes: 512 KB
    Kafka:
        Brokers:
            - kafka0:9092
            - kafka1:9092
            - kafka2:9092
            - kafka3:9092
    Organizations:

Application: &ApplicationDefaults                  ⑥
    Organizations:

Capabilities:                                      ⑦
    Global: &ChannelCapabilities
        V1_1: true
    Orderer: &OrdererCapabilities
        V1_1: true
    Application: &ApplicationCapabilities
        V1_1: true
```

对该文件的内容解释如下。

①用于定义 Orderer 自己的系统通道的创世区块信息，包含一个 Orderer 组织（OrdererOrg）并定义了一个联盟（SampleConsortium）。

②用于定义一个名为 TwoOrgsChannel 的业务通道，在这个通道里使用了 SampleConsortium，需要注意，定义联盟时的 Key 为 Consortiums，而使用时的 Key 为 Consortium，在这个业务通道里包含一个应用（Application），并包含 Org1 与 Org2 这两个组织，这里没有另外定义 Orderer 组织，因此会使用默认的 Orderer 组织。

③是对 Orderer 组织的定义，主要包含组织名称（Name）、MSP ID（ID）及 MSP 加密材料目录（MSPDir）。组织名称需要与之前定义的名称一致，可以与 MSP ID 不同；MSP ID 是身份的代表，需要与 MSP 目录匹配，否则会造成签名校验失败。MSP 目录则直接指向 cryptogen 所生成目录中的子目录之一，我们可以看到所有节点的 MSP 目录都被放在 crypto-config 目录下，在正式环境下需要将每个节点自己的 MSP 目录隔离开来，避免有的节点可以读取到非自身节点的私钥，从而伪造签名。

④是对 Peer 组织的定义，除了包含 Peer 组织名称（Name）、MSP ID（ID）、MSP 目录（MSPDir），还包含一个 AnchorPeer，也就是锚定节点，AnchorPeer 是组织之间的通信入口，当然 AnchorPeer 并不是固定的，在示例的步骤 o 和 p 中就对两个组织的 AnchorPeer 进行了更新。

⑤是对 Orderer 的初始化，定义 Orderer 的类型（Type）、Orderer 地址（Addresses）、打包时长（BatchTimeout）、打包大小（BatchSize）、Kafka 集群（Kafka.Brokers）、排序节点组织（Organizations）等信息；其中 Orderer 类型主要有 Solo（单节点）与 Kafka（集群）两种模式；Orderer 地址可以是一个列表，在这个列表中的所有排序节点都能接收交易提案；打包时长指的是在接收一个交易提案后开始计时，最长允许多长时间内的交易提案打包形成一个区块；打包大小则指的是在这段时间内交易提案满足最大个数（MaxMessageCount）、总大小（PreferedMaxBytes）时，就会将交易提案打包生成区块，其中 PreferedMaxBytes 关系到交易切分规则；需要注意的是，当交易提案大于绝对大小 AbsoluteMaxBytes 时，会被拒绝。

⑥是对应用的定义，主要定义在应用中包含的组织（Organizations），应用只应出现在业务通道中，不应出现在系统通道定义中。

⑦是对能力的定义，主要是定义网络的兼容性，当某个版本的功能开启时，不兼容此

版本的节点便不能参与到网络中，当前定义的是 V1_1，也就是 Fabric 1.1 版本的功能，如果此时有低于该版本的 Peer 节点试图加入网络中，则会报错，以防止因为版本差异导致网络内部信息校验、交易提案或者生成区块不一致。

3.3.2 Orderer节点的详细配置与定义

在使用这两个配置文件设置了网络结构之后，会生成 orderer.block 文件作为创世区块，所有的网络初始配置信息都会被写入创世区块中。步骤 e 会启动 Orderer 节点，在启动时会根据参数加载 orderer.block 创世区块；Orderer 节点在启动时首先会加载容器内自带的配置文件 orderer.yaml，可以在 Fabric 源码的 sampleconfig 目录下找到这个文件，该文件主要包含通用（General）、账本（FileLedger、RAMLedger）及消息队列（Kafka）三方面的设置，如下所述。

通用设置如下。

```
General:

    # 区块保存方式，有以下三种可选：
    #  - file，以二进制文件方式保存，为默认的保存方式
    #  - ram，只存在于内存中，在程序结束后就被释放掉
    #  - json，以JSON格式保存区块，但被序列化的部分不会自动反序列化
    # 需要注意的是，在生产环境下只能使用file这种保存方式
    LedgerType: file

    # Orderer节点的监听地址
    ListenAddress: 127.0.0.1

    # Orderer节点的监听端口
    ListenPort: 7050

    # gRPC服务端的TLS设置
    TLS:
        Enabled: false
        PrivateKey: tls/server.key
        Certificate: tls/server.crt
        RootCAs:
          - tls/ca.crt
        ClientAuthRequired: false
        ClientRootCAs:
```

```
    # gRPC服务端的心跳设置
    Keepalive:
        # ServerMinInterval定义了客户端ping包的最小间隔，如果间隔小于设定值，
        # 服务器就会断开客户端
        ServerMinInterval: 60s
        # Server向Client发送心跳的间隔时间
        ServerInterval: 7200s
        # ServerTimeout是服务器等待客户端响应的时长，如果超时未响应，则会断开连接
        ServerTimeout: 20s

    # 日志级别，主要包含CRITICAL、ERROR、WARNING、NOTICE、INFO和DEBUG 这6个级别，在调试时
    # 建议用DEBUG

    LogLevel: info

    # 日志格式，在一些没有颜色输出的终端上可以去掉与颜色相关的配置
    LogFormat: '%{color}%{time:2006-01-02 15:04:05.000 MST} [%{module}] %{shortfunc}
-> %{level:.4s} %{id:03x}%{color:reset} %{message}'

    # 创世区块的生成方式，用于指定在Orderer节点自举时系统通道的创世区块来源方式
    # 包含provisional和file选项
    #   - provisional: 通过 GenesisProfile 指定的配置动态生成创世区块
    #   - file: 通过GenesisFile指定的文件加载为创世区块
    GenesisMethod: provisional

    # 指定configtx.yaml文件中的创世区块配置
    GenesisProfile: SampleInsecureSolo

    # 指定创世区块文件
    GenesisFile: genesisblock

    # 指定MSP加密材料的路径
    LocalMSPDir: msp

    # 指定本地的MSPID，默认为DEFAULT
    LocalMSPID: DEFAULT

    # 基于pprof的Go语言性能测量工具
    # 可访问golang.org网站来查看相关文档
    Profile:
```

```
        Enabled: false
        Address: 0.0.0.0:6060

    # BCCSP 用于配置区块链加密服务提供者
    BCCSP:
        # Default用于指定区块链使用的加密服务，有以下两个选项
        #  - SW: 软件加密服务提供程序
        #  - PKCS11: CA硬件加密服务提供程序
        # 如果首选加密服务不能使用，则将默认使用软件加密服务提供程序
        Default: SW

        # SW 加密选项
        SW:
            # 用于指定哈希算法与安全级别，因为SHA2在BCCSP之外多处直接使用，
            # 因此这部分将来需要重构成完全可配置
            Hash: SHA2
            Security: 256
            # 密钥库的存储位置，默认位置是'LocalMSPDir'/keystore
            # 也就是msp/keystore目录下
            FileKeyStore:
                KeyStore:

    # Authentication包含了与认证相关的配置参数
    Authentication:
        # TimeWindow定义了服务器可以接受的客户端时间差异，
        # 也就是客户端提交时的时间戳与服务器相差不能超过指定的值
        TimeWindow: 15m
```

账本设置如下。

```
FileLedger:
    # Location指定区块的存储目录，若未指定，则会在每次启动时创建Prefix指定的临时目录
    Location: /var/hyperledger/production/orderer
    Prefix: hyperledger-fabric-ordererledger

# 与内存账本相关的配置
RAMLedger:

    # HistorySize 定义了内存账本保存的区块最大数量，如果超出，则会丢掉老区块
    HistorySize: 1000
```

消息队列集群设置如下。

```
Kafka:

    # 定义与Kafka交互的重试规则
    Retry:
        # 当创建新通道或者重新加载已有通道时，Orderer节点会按照以下步骤与Kafka集群交互
        # (1) 为对应通道的Kafka分区创建一个生产者（writer）
        # (2) 使用该生产者发送一个 CONNECT 消息到对应的分区
        # (3) 为对应通道的Kafka分区创建一个消费者（reader）
        # 如果上述步骤失败，则会根据下列设置进行重试，在此期间Orderer节点无法对此通道进行读写，
        # 直到上述步骤都成功
        ShortInterval: 5s
        ShortTotal: 10m
        LongInterval: 5m
        LongTotal: 12h
        # 网络超时设置
        NetworkTimeouts:
            DialTimeout: 10s
            ReadTimeout: 10s
            WriteTimeout: 10s
        # Kafka集群选主期间的Metadata请求的重试规则
        Metadata:
            RetryBackoff: 250ms
            RetryMax: 3
        # 写入Kafka集群时的重试规则
        Producer:
            RetryBackoff: 100ms
            RetryMax: 3
        # 读取Kafka集群时的重试规则
        Consumer:
            RetryBackoff: 2s

    # Verbose: 是否开启Kafka集群交互日志
    Verbose: false

    # TLS用于设置Orderer节点与Kafka之间的通信加密设置，与gRPC的TLS是有明显区分的，
    # 为了防止Peer节点伪造或者得到Orderer节点共识过程中的信息
    TLS:
      Enabled: false
      PrivateKey:
```

```
        Certificate:
        RootCAs:

      # Version 用于定义Kafka集群协议版本号，默认是0.10.2.0
      Version:

  Debug:
      # BroadcastTraceDir用于将广播服务的请求写入指定的目录中
      BroadcastTraceDir:

      # DeliverTraceDir 用于将Deliver服务的请求写入指定的目录中
      DeliverTraceDir:
```

在容器内启动时，我们通过环境变量覆盖了一些原有设置：

```
  # 指定默认的配置文件路径
  -e ORDERER_CFG_PATH=/shared/
  # 覆盖 General.LedgerType，设置为file类型
  -e ORDERER_GENERAL_LEDGERTYPE=file \
  # 覆盖 General.ListenPort，设置为端口31010
  -e ORDERER_GENERAL_LISTENPORT=31010 \
  # 覆盖 General.ListenAddress，设置为监听所有IP
  -e ORDERER_GENERAL_LISTENADDRESS=0.0.0.0 \
  # 覆盖 General.LocalMSPDir，设置到指定的目录下
  -e ORDERER_GENERAL_LOCALMSPDIR=/shared/crypto-config/ordererOrganizations/
example.com/orderers/orderer.example.com/msp \
  # 覆盖 FileLedger.Location，设置到指定的目录/mnt/ledger/orderer1下
  -e ORDERER_FILELEDGER_LOCATION=/mnt/ledger/orderer1 \
  # 覆盖 General.BatchTimeout，设置为最长1秒
  -e ORDERER_GENERAL_BATCHTIMEOUT=1s \
  # 覆盖 General.OrdererType，设置为Solo模式
  -e ORDERER_GENERAL_ORDERERTYPE=solo \
  # 覆盖 General.LogLevel，设置为DEBUG级别
  -e ORDERER_GENERAL_LOGLEVEL=debug \
  # 覆盖 General.LocalMSPID，设置为OrdererMSP
  -e ORDERER_GENERAL_LOCALMSPID=OrdererMSP \
  # 覆盖 General.GenesisFile，设置到指定的文件下
  -e ORDERER_GENERAL_GENESISFILE=/shared/orderer.block \
  # 覆盖 General.GenesisMethod，设置类型为file，加载指定的创世区块文件
  -e ORDERER_GENERAL_GENESISMETHOD=file \
  # 覆盖 General.GenesisProfile的设置
```

```
-e ORDERER_GENERAL_GENESISPROFILE=initial \
# 覆盖 General.TLS.Enable的设置，不启用TLS加密
-e ORDERER_GENERAL_TLS_ENABLED=false \
```

在进行环境设置之后，我们通过命令启动 Orderer 节点，orderer 命令的参数及其作用如下。

◎　help [<command>...]：显示帮助。
◎　start*：启动 Orderer 节点。
◎　Version：显示版本号。
◎　Benchmark：运行 Orderer 节点的 Benchmark 测试。

在启动 Orderer 节点后，我们将会启动第 1 个组织中的第 1 个节点 org1peer1，Peer 节点的运行时默认配置主要被放在 core.yaml 文件中。可以看到，在 core.yaml 中定义了非常多的内容，包括 Logging、Peer、VM、Chaincode、Ledger 等，涵盖了系统中的大部分流程与环节，因此了解这些参数对于理解 Fabric 的运行机制与原理很有帮助。

3.3.3　Peer 节点的详细配置与定义

Peer 节点的配置都在 core.yaml 文件中，可以在 sampleconfig 目录下找到 core.yaml 文件。该文件的内容包括 Peer 节点设置(peer)、虚拟机设置(vm)、Chaincode 设置(chaincode)、账本设置（ ledger)、日志设置（ logging ）等。由于 Peer 设置较多，所以下面对该文件进行分段介绍。

Peer 节点设置中的基础配置如下：

```
peer:
    # 当前节点的ID
    id: jdoe

    # 逻辑分割网络ID
    networkId: dev

    # 默认的监听地址，监听所有IP
    listenAddress: 0.0.0.0:7051

    # Peer节点被用于监听Chaincode的地址，在Fabric 1.1之后将Chaincode的监听地址分离出来，
    # 避免其他节点拦截或者伪造Chaincode通信，如果没有设置，
    # 则默认是peer.address设置的IP下的端口7052
```

```
    # chaincodeListenAddress: 0.0.0.0:7052

    # Chaincode的地址，如果没有设置，则默认是listenAddress
    # chaincodeAddress: 0.0.0.0:7052

    # 在作为Peer节点时，该地址用于指定相同组织下的其他节点地址；
    # 在作为CLI客户端时，该地址用于指定目标的Peer节点
    address: 0.0.0.0:7051

    # 如果运行在Docker容器中，则开启此参数可以自动检测IP地址
    addressAutoDetect: false

    # 设置Golang的最大进程数，一般是内核数，如果n<1，则会保持现有设置
    gomaxprocs: -1

    # Peer节点与客户端的心跳设置
    keepalive:
        # 节点允许客户端心跳的最小间隔，如果客户端发送的心跳间隔小于该值，
        # 则Peer节点将会断开连接
        minInterval: 60s
        # 客户端心跳设置
        client:
            # 客户端心跳间隔，必须大于或者等于peer.minInterval
            interval: 60s
            # 客户端与服务器的连接超时设置
            timeout: 20s
        # 与Orderer节点之间的心跳设置
        deliveryClient:
            # 心跳间隔，必须大于或等于指定的Orderer节点
            # orderer.General.Keepalive.ServerMinIntervals
            interval: 60s
            # 与Orderer连接的超时设置
            timeout: 20s
```

Peer 节点设置中的 Gossip 设置如下：

```
    gossip:
        # Gossip在初始化时连接的地址集合，注意这些节点都要在统一的组织中
        bootstrap: 127.0.0.1:7051

        # 注意：orgLeader与useLeaderElection参数是互斥的，
```

```
# 将两者都设为true会导致节点不能启动，如果同一组织内的Peer节点都
# 设置了useLeaderElection=false，则请确保在组织内至少有一个orgLeader

# 设置是否使用自动选主，同一组织中的主Peer节点用于与Orderer节点通信并获取区块
# 对大型网络建议使用自动选主
useLeaderElection: true
# 静态定义节点是否为组织主节点（leader），主节点用于维持与Orderer节点的链接，
# 并将获取到的区块在所在的组织内传播
orgLeader: false

# 定义Peer节点公开给同组织内其他节点的地址
endpoint:
# 在内存中保存的最大区块数
maxBlockCountToStore: 100
# 连续消息推送之间的最长时间(单位为毫秒)
maxPropagationBurstLatency: 10ms
# 消息存储的最大数量，在达到这个数量时触发推送给其他Peer节点
maxPropagationBurstSize: 10
# 同一个消息向其他Peer节点发送的次数
propagateIterations: 1
# 一次推送多少个Peer节点
propagatePeerNum: 3
# 拉取的时间间隔(单位为秒)
pullInterval: 4s
# 从多少个Peer节点去拉取
pullPeerNum: 3
# 从Peer节点拉取状态消息的频率(单位为秒)
requestStateInfoInterval: 4s
# 向Peer节点推送状态消息的频率(单位为秒)
publishStateInfoInterval: 4s
# stateInfo消息的有效期限
stateInfoRetentionInterval:
# 设置多长时间内的心跳消息包含自身身份证明(单位为秒)
publishCertPeriod: 10s
# 是否忽略区块验证消息(当前未使用)
skipBlockVerification: false
# 建立连接超时时间(单位为秒)
dialTimeout: 3s
# 连接超时时间(单位为秒)
connTimeout: 2s
# 接收消息的缓冲大小
```

```
    recvBuffSize: 20
    # 发送消息的缓冲大小
    sendBuffSize: 200
    # 在拉取后等待多久开始处理传入摘要（单位为秒）
    digestWaitTime: 1s
    # 在拉取引擎在收到Hello消息后等待多久移除这个消息中的随机数(单位为秒)
    requestWaitTime: 1s
    # 处理传入摘要后多久结束拉取(单位为秒)
    responseWaitTime: 2s
    # 心跳消息的间隔时间(单位为秒)
    aliveTimeInterval: 5s
    # 心跳的超时时间(单位为秒)
    aliveExpirationTimeout: 25s
    # 重试间隔(单位为秒)
    reconnectInterval: 25s
    # 用于定义对外部组织暴露的地址，如果不设置则其他组织不会知晓当前节点
    externalEndpoint:
    # 选主服务配置
    election:
        # 在选主服务启动后成员视图稳定下来的最长时间(单位为秒)
        startupGracePeriod: 15s
        # Gossip成员关系的稳定性采样间隔时间(单位为秒)
        membershipSampleInterval: 1s
        # Leader节点的心跳检查周期(单位为秒)
        leaderAliveThreshold: 10s
        # 从Peer节点提交选主提案到发送消息声明自己是Leader节点的时间，
        # 一般是leaderAliveThreshold的一半(单位为秒)
        leaderElectionDuration: 5s

    # 在同步区块时，已接收到但还未提交的区块数据被称为私有数据（Private Data，pvtData），
    # 会被存储在临时商店（TrasientStore）中
    pvtData:
        # 如果在同步区块时发现区块数据不全，则会拉取丢失的部分数据，
        # pullRetryThreshold定义了拉取丢失数据操作允许的时长
        pullRetryThreshold: 60s
        # transientstoreMaxBlockRetention定义了私有数据与当前区块的高度，当提交的区块高
        # 度是transientstoreMaxBlockRetention的整数倍时，早期的私有数据会被清除
        transientstoreMaxBlockRetention: 1000
        # 在背书环节传播私有数据时等待Peer节点确认的超时时间
        pushAckTimeout: 3s
```

Peer 节点设置中与 EventHub 及 TLS 相关的设置如下：

```
events:
    # EventHub的地址
    address: 0.0.0.0:7053

    # EventChannel的缓存大小
    buffersize: 100

    # 虽然名为超时，但实际上承载了一些超时之外的逻辑，
    # 在事件发送时先判断Timeout的值，然后根据eventChannel的缓存状态进行对应的逻辑处理：
    # timeout < 0，如果缓存满了，就中断且不发送
    # timeout == 0，如果缓存满了，就阻塞到事件发送出去
    # timeout > 0，如果缓存满了，就会阻塞到Timeout
    timeout: 10ms

    # 在validateEventMessage验证事件消息时会检查事件时间戳与当前
    # 系统时间相差是否超过TimeWindow时间，如果超过验证则会失败
    timewindow: 15m

    # 事件服务器与客户端之间的心跳设置
    keepalive:
        # 客户端发送心跳的最小间隔时间，如果少于这个时间间隔，则会断开连接
        minInterval: 60s

# TLS 设置
tls:
    enabled:  false
    clientAuthRequired: false
    cert:
        file: tls/server.crt
    key:
        file: tls/server.key
    rootcert:
        file: tls/ca.crt
    clientRootCAs:
        files:
          - tls/ca.crt
    clientKey:
        file:
    clientCert:
```

```
        file:
```

Peer 节点设置中与身份认证、MSP、Handlers 相关的设置如下：

```
authentication:
    # TimeWindow定义了服务器可以接受的客户端时间差异，也就是客户端
    # 提交时的时间戳与服务器相差不能超过指定的值
    timewindow: 15m

# Peer节点用于存放账本与状态数据的路径，注意该路径必须受到访问控制保护
# 避免破坏Peer节点操作导致的其他修改
fileSystemPath: /var/hyperledger/production

# BCCSP 设置需要与Orderer节点的位置相同
BCCSP:
    Default: SW
    SW:
        Hash: SHA2
        Security: 256
        FileKeyStore:
            KeyStore:

# MSP加密材料的存放路径
mspConfigPath: msp

# 本地MSPID，注意需要修改默认值
localMspId: DEFAULT
```

与 Delivery 服务相关的设置如下：

```
deliveryclient:
    # 设置Delivery服务重连所花费的时间
    reconnectTotalTimeThreshold: 3600s

# 本地MSP类型，默认是bccsp
localMspType: bccsp
```

性能测量工具设置如下：

```
profile:
    enabled:    false
    listenAddress: 0.0.0.0:6060

# Handlers自定义处理程序，可以过滤和改变在Peer节点内传递的对象
```

```
    # 例如:
    #    Auth filter - 拒绝或转发来自客户端的提案
    #    Decorators  - 修饰Chaincode的输入
    # 有效的Chaincode输入包括:
    #   - 静态编译过滤器需要一个定义在 core/handlers/library/library.go中的工厂方法名称
    #   - 可插拔过滤器需要定义指向.so的路径，如果有.so文件，则不会再调用静态编译
    # Auth filters与Decorators按照定义顺序执行，例如:
    # authFilters:
    #   -
    #     name: FilterOne
    #     library: /opt/lib/filter.so
    #   -
    #     name: FilterTwo
    # decorators:
    #   -
    #     name: DecoratorOne
    #   -
    #     name: DecoratorTwo
    #     library: /opt/lib/decorator.so
    handlers:
       authFilters:
         -
           name: DefaultAuth
         -
           name: ExpirationCheck    # 该过滤器检查X509证书是否过期
       decorators:
         -
           name: DefaultDecorator

    # 并行执行交易验证的goroutines数，默认是runtime.NumCPU()的数量
    validatorPoolSize:
```

虚拟机设置如下：

```
vm:

    # 虚拟机管理系统的Endpoint，Docker一般用下列之一:
    # unix:///var/run/docker.sock
    # http://localhost:2375
    # https://localhost:2376
    endpoint: unix:///var/run/docker.sock
```

```
    # Docker配置，Fabric使用Docker API创建Chaincode容器，Docker API参考：
    # docs.docker.com网站下的v1.37版API
    docker:
       tls:
          enabled: false
          ca:
             file: docker/ca.crt
          cert:
             file: docker/tls.crt
          key:
             file: docker/tls.key

                   # 定义Chaincode容器是否使用 stdout/err 输出，调试用
       attachStdout: false

                   # 创建Docker容器的参数，使用集群的ipam与dns-server可以高效地创建容器
                   # NetworkMode - 设置容器的网络模式，对应Docker API中的
                   # NetworkMode参数，默认是 host参数，还支持bridge、none、
                   # container:<name|id>及自定义网络名称
                   # Dns - 容器使用的DNS列表
                   # 注意：出于安全考虑，'Privileged'、 'Binds'、 'Links' 与
                   # 'PortBindings'属性不支持自定义
                   # LogConfig - 设置日志级别，不支持环境变量中设置的LogConfig，更多信息
                   # 请参考docs.docker.com网站下的logging
       hostConfig:
          NetworkMode: host
          Dns:
            # - 192.168.0.1
          LogConfig:
             Type: json-file
             Config:
                max-size: "50m"
                max-file: "5"
          Memory: 2147483648
```

Chaincode 设置如下：

```
chaincode:

    # ID主要用于向Peer节点注册Chaincode，通常包含名字（Name）、路径（Path）与版本号（Version）
    # 通常通过peer chaincode install命令的参数来设置
    id:
```

```
    path:
    name:

# Chaincode编译环境镜像
builder: $(DOCKER_NS)/fabric-ccenv:$(ARCH)-$(PROJECT_VERSION)

# 在Chaincode实例化时是否强制拉取Docker基础镜像，用于使用镜像标签（例如latest）的情况
pull: false

golang:
    # Golang基础镜像只需要fabric-baseos即可满足
    runtime: $(BASE_DOCKER_NS)/fabric-baseos:$(ARCH)-$(BASE_VERSION)

    # 是否应该动态链接Golang Chaincode
    dynamicLink: false

car:
    # 随着平台目录的扩大，car将可能需要更多的设施（JVM等）
    # 现在我们只需要fabric-baseos
    runtime: $(BASE_DOCKER_NS)/fabric-baseos:$(ARCH)-$(BASE_VERSION)

java:
    # 该镜像基于java:openjdk-8带上附加Shim层打包的编译工具
    # 该镜像包含了Java Chaincode运行需要的Shim层库文件
    Dockerfile:  |
        from $(DOCKER_NS)/fabric-javaenv:$(ARCH)-$(PROJECT_VERSION)

node:
    # Node.js Chaincode在运行时需要Node.js引擎，所以引用了baseimage镜像而非baseos
    runtime: $(BASE_DOCKER_NS)/fabric-baseimage:$(ARCH)-$(BASE_VERSION)

# 启动容器等待注册通过的超时时长，对于单元测试来说1秒应该足够了
# 需要注意的是，在实际运行环境下有可能因为pull镜像造成时长增加
startuptimeout: 300s

# Chaincode初始化与调用操作的超时时间设置，该设置影响所有通道的所有Chaincode
# 包括系统Chaincode。需要注意的是，如果在调用期间Chaincode失效（例如在调试环境下被清除）
# Peer节点会自动构建Chaincode镜像，可能会耗费更多的时间；生产环境的Chaincode基本不会被
# 删除，所以时长可以适当缩短
executetimeout: 30s
```

```
    # Chaincode有两种运行模式：dev模式与net模式，在dev模式下，Chaincode需要用户在本机上通过
    # 命令行手动启动，在net模式下Peer节点会自动在Docker容器中启动Chaincode
    mode: net

    # Peer节点与Chaincode之间的心跳间隔，如果两者之间的通信需要穿过一个不支持keep-alive的代理，
    # 则该参数可以用来维持链接。当值不大于0时关闭keepalive
    keepalive: 0

    # 系统Chaincode启用的名单。如果添加了myscc的系统Chaincode，则需要在下列名单中增加
    # "myscc:enable"，并在chaincode/importsysccs.go中注册
    system:
        cscc: enable
        lscc: enable
        escc: enable
        vscc: enable
        qscc: enable

    # 系统Chaincode插件：除了通过core/chaincode/importsysccs.go导入并编译，
    # 系统链接代码还可以被编译为Go插件的.so文件进行加载。
    # 有关示例请参阅examples/plugins/scc。像常规系统链接代码一样，
    # 插件也必须在上面的chaincode.system部分列出
    systemPlugins:
            # 示例配置：
            # - enabled: true
            #   name: myscc
            #   path: /opt/lib/myscc.so
            #   invokableExternal: true
            #   invokableCC2CC: true

    # Chaincode的日志级别
    logging:
      level:  info
      shim:   warning
      format: '%{color}%{time:2006-01-02 15:04:05.000 MST} [%{module}] %{shortfunc}
-> %{level:.4s} %{id:03x}%{color:reset} %{message}'
```

账本设置如下：

```
ledger:

    blockchain:
```

```
    # 状态数据库
    state:
        # stateDatabase - 可以选择 "goleveldb"或者"CouchDB"
        # goleveldb - 使用goleveldb作为状态数据库，为默认的选项
        # CouchDB - 使用CouchDB作为状态数据库
        stateDatabase: goleveldb
        couchDBConfig:
            # 建议将CouchDB在Peer节点本地运行，不要在Docker-compose中将
            # CouchDB容器的端口映射到服务器端口上，否则需要将两者之间的通信适当加密来保证安全性
            couchDBAddress: 127.0.0.1:5984
            # CouchDB用户名
            username:
            # 建议通过环境变量传入密码
            # (eg LEDGER_COUCHDBCONFIG_PASSWORD).
            # 如果密码写在这里，那么此文件需要受到权限控制保护，避免被其他用户获取
            password:
            # 遇到CouchDB错误时的重试次数
            maxRetries: 3
            # 在Peer节点启动时遇到CouchDB错误时的重试次数
            maxRetriesOnStartup: 10
            # 请求超时时间
            requestTimeout: 35s
            # 每次查询返回的最大记录数
            queryLimit: 10000
            # CouchDB批量更新的最大记录数
            maxBatchUpdateSize: 1000
            # 在每N个区块之后预热索引，在值为1时每个区块提交后都会进行预热索引来加速查询，
            # 但可能影响写入效率；值越大会写入效率越高，但可能延长查询响应时间，
            # 这个值的设置需要根据具体的使用场景进行权衡
            warmIndexesAfterNBlocks: 1

    history:
        # 在交易中，所有的键值变化的历史都会被记录在HistoryDatabase中
        # HistoryDatabase使用goleveldb作为存储，通过GetHistoryForKey函数
        # 可以查询到指定Key的更新历史，enableHistoryDatabase指定是否启用HistoryDatabase
        enableHistoryDatabase: true
```

日志设置如下：

```
logging:
    # 默认的日志级别
    level:      info
```

```
    # 每个模块的日志级别
    cauthdsl:  warning
    gossip:    warning
    grpc:      error
    ledger:    info
    msp:       warning
    policies:  warning
    peer:
    gossip:    warning
    # 日志格式定义
    format: '%{color}%{time:2006-01-02 15:04:05.000 MST} [%{module}] %{shortfunc}
-> %{level:.4s} %{id:03x}%{color:reset} %{message}'
```

Peer 节点在启动后加载了 core.yaml 中的配置内容，我们通过环境变量覆盖了部分定义：

```
  # 指定LocalMSPID为Org1MSP，需要对应2orgs-configtx.yaml中的定义
  -e CORE_PEER_LOCALMSPID=Org1MSP
  # 指定MSPConfigPath本地加密材料目录
  -e CORE_PEER_MSPCONFIGPATH=/shared/crypto-config/peerOrganizations/org1.example.
com/peers/peer0.org1.example.com/msp/
  # 指定不启用Gossip网络自动选主
  -e CORE_PEER_GOSSIP_USELEADERELECTION=false
  # 指定当前节点为Gossip网络当前组织的Ledger
  -e CORE_PEER_GOSSIP_ORGLEADER=true
  # Peer节点的监听地址
  -e CORE_PEER_LISTENADDRESS=$INTERNAL_IP:5010
  # EventHub的监听地址
  -e CORE_PEER_EVENTS_ADDRESS=0.0.0.0:5011
  # Gossip初始化时的连接地址
  -e CORE_PEER_GOSSIP_BOOTSTRAP=$INTERNAL_IP:5010
  # Peer的逻辑网络ID
  -e CORE_PEER_NETWORKID=nid1
  # VM管理Endpoint
  -e CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
  # 启用节点Committer
  -e CORE_PEER_COMMITTER_ENABLED=true
  # 关闭TLS加密
  -e CORE_PEER_TLS_ENABLED=false
  # 关闭性能测量功能
  -e CORE_PEER_PROFILE_ENABLED=false
  # 状态数据库使用goleveldb
```


```
-e CORE_LEDGER_STATE_STATEDATABASE=goleveldb
# 指定配置文件所在的地址，包括core.yaml配置
-e FABRIC_CFG_PATH=/etc/hyperledger/fabric/
# 指定日志级别为DEBUG
-e CORE_LOGGING_LEVE=debug
# 指定Gossip日志级别为DEBUG
-e CORE_LOGGING_GOSSIP=debug
# 指定Chaincode启动的超时时间为600秒
-e CORE_CHAINCODE_STARTUPTIMEOUT=600s
```

3.3.4　peer 命令

在设置完这些环境变量后，我们通过 peer 命令启动 Peer 节点；在 Fabric 中 Peer 有两种角色：一种是 Peer 节点服务，包括 Endorser 与 Committer；一种是 CLI 客户端，与 Peer 节点或者 Orderer 节点通信，执行特定的功能，用于创建通道、部署 Chaincode、提交交易、查询交易等，因此 peer 命令的用法较为复杂。下面介绍 peer 命令的主要参数，并解读其对应的功能。

（1）--help, -h：显示帮助。

（2）--logging-level：设置当前 peer 命令的日志级别，主要包含 CRITICAL、ERROR、WARNING、NOTICE、INFO 及 DEBUG 这 6 个级别。

（3）--test.coverprofile：将代码覆盖率输出到指定的文件中，默认是 coverage.cov。

（4）--version, -v：显示当前 Peer 节点的版本。

除此之外，peer 命令还包含 peer chaincode（链码操作）、peer channel（通道操作）、peer logging（日志操作）、peer node（服务节点操作）等子命令。接下来，我们分别介绍这些子命令的参数。

peer chaincode 子命令的参数如下。

（1）--cafile：指定排序节点的 PEM 编码信任证书文件的路径。

（2）--help, -h：显示帮助。

（3）--orderer, -o：指定排序节点的服务器地址与端口。

（4）--tls：指定在访问排序节点时是否使用 TLS。

（5）install：将指定的 Chaincode 打包为 ChaincodeDeploymentSpec 并保存到指定的节点，包含以下参数。

◎ --ctor, -c：JSON 格式的 Chaincode 参数，默认是{}。
◎ --lang, -l：Chaincode 编码语言，默认是 Golang。
◎ --name, -n：Chaincode 的名称。
◎ --path, -p：在 install/instantiate/upgrade 时指定 Chaincode 的版本。
◎ --version, -v：在 install/instantiate/upgrade 时指定 Chaincode 的版本。

（6）instantiate：将 Chaincode 实例化部署到网络上，包含以下参数。

◎ --channelID, -C：指定通道 ID，默认是 testchainid。
◎ --ctor, -c：JSON 格式的 Chaincode 参数，默认是{}。
◎ --escc, -E：Chaincode 的 ESCC 名称。
◎ --lang, -l：Chaincode 的编码语言，默认是 Golang。
◎ --name, -n：Chaincode 的名称。
◎ --policy, -P：Chaincode 的背书策略。
◎ --version, -v：在 install/instantiate/upgrade 时指定 Chaincode 的版本。
◎ --vscc, -V：Chaincode 的 VSCC 名称。

（7）invoke：调用指定的 Chaincode，会尝试将已背书的交易提交到网络中，包含以下参数。

◎ --channelID, -C：指定通道 ID，默认是 testchainid。
◎ --ctor, -c：JSON 格式的 Chaincode 参数，默认是{}。
◎ --name, -n：Chaincode 的名称。

（8）package：在本地将 Chaincode 打包为 ChaincodeDeploymentSpec，包含以下参数。

◎ --cc-package, -s：指定创建为原始格式或者多个所有者可以签名的格式，默认是原始格式，在带-s 参数后也需要加上--sign 参数才能生效。
◎ --ctor, -c：JSON 格式的 Chaincode 参数，默认是{}。
◎ --instantiate-policy, -i：Chaincode 打包的实例化策略。
◎ --lang, -l：Chaincode 的编码语言，默认是 Golang。
◎ --name, -n：Chaincode 的名称。
◎ --path, -p：Chaincode 文件的存放路径。

◎　--sign, -S：签名打包文件。

◎　--version, -v：在 install/instantiate/upgrade 时指定 Chaincode 的版本。

（9）query：调用 Chaincode 进行查询，与 invoke 不同的是，交易不会被提交到网络中，包含以下参数。

◎　--channelID, -C：指定通道 ID，默认是 testchainid。

◎　--ctor, -c：JSON 格式的 Chaincode 参数，默认是{}。

◎　--hex, -x：以十六进制字节数组方式输出查询结果，与--raw 不兼容。

◎　--name, -n：Chaincode 的名称。

◎　--raw, -r：以原始字节方式输出查询结果，否则以可打印字符方式输出。

◎　--tid, -t：指定一个 ID 生成算法，这个算法包含哈希函数与解码算法两部分，默认是 sha256base64。

（10）signpackage：签名指定的 Chaincode 包。

（11）upgrade：将已有的 Chaincode 升级为指定的 Chaincode。在交易提交成功后新的 Chaincode 会马上生效，包含以下参数。

◎　--channelID, -C：指定通道 ID，默认是 testchainid。

◎　--ctor, -c：JSON 格式的 Chaincode 参数，默认是{}。

◎　--escc, -E：Chaincode 的 ESCC 名称。

◎　--lang, -l：Chaincode 的编码语言，默认是 Golang。

◎　--name, -n：Chaincode 的名称。

◎　--path, -p：Chaincode 文件的存放路径。

◎　--policy, -P：Chaincode 的背书策略。

◎　--version, -v：在 install/instantiate/upgrade 时指定 Chaincode 的版本。

◎　--vscc, -V：Chaincode 的 VSCC 名称。

peer channel 子命令用于操作通道，注意以下“指定的节点”一般是由配置文件中的 peer.address 或者环境变量 CORE_PEER_ADDRESS 指定的。peer channel 子命令的参数如下。

（1）--cafile：指定排序节点的 PEM 编码信任证书文件的路径。

（2）--orderer, -o：指定排序节点的服务器地址与端口。

（3）--tls：指定在访问排序节点时是否使用 TLS。

（4）create：创建通道，包含以下参数。

◎ --channelID, -c：指定通道 ID，默认是 testchainid。

◎ --file, -f：由工具（例如 configtxgen）生成的配置交易文件，用于提交给排序节点。

◎ --timeout, -t：指定超时时间，默认为 5 秒。

（5）fetch：获取指定的通道所指定的区块，并将区块保存到参数指定的文件中，包含以下参数。

◎ newest：获取最新的一个区块。

◎ oldest：获取最老的一个区块，一般是创世区块。

◎ config：获取最新的配置区块。

◎ (number)：根据 block number 获取区块。

（6）join：将指定的节点加入指定的通道中，包含以下参数。

◎ --blockpath, -b：指定创始区块文件。

（7）list：查询指定的节点所加入的通道，注意同一个节点可以加入多个通道中，如下所示。

```
[root@localhost fabric]# peer channel list
2018-02-21 23:24:46.654 CST [channelCmd] list -> INFO 002
Channels peers has joined to:
2018-02-21 23:24:46.654 CST [channelCmd] list -> INFO 003
ch1
```

（8）update：签名由工具生成的更新交易文件并提交给通道，-f、-o 和-c 是必需的参数。

◎ --file, -f：由工具（例如 configtxgen）生成的配置交易文件，用于提交给排序节点。

◎ -orderer, -o：指定排序节点的服务器地址与端口。

◎ --channelID, -c：指定通道 ID，默认是 testchainid。

peer logging 子命令用于获取或者设置指定的节点的日志级别，包含以下参数。

（1）getlevel：获取指定的节点所指定的模块的日志级别，需要包含模块的名称，如下所示。

```
[root@localhost fabric]# peer logging getlevel msp
2018-02-21 23:32:42.601 CST [cli/logging] getLevel -> …
Current log level for peer module 'msp': WARNING
```

（2）revertlevels：将日志级别退回到启动时的设置。

（3）setlevel：设置指定的节点所指定的模块的日志级别，模块名称可以是正则表达式。

```
[root@localhost fabric]# peer logging setlevel '\bm\w+' INFO
…
Log level set for peer modules matching regular expression '\bm\w+': INFO
```

peer node 子命令为 Peer 节点的服务端命令，包含以下参数。

（1）start：启动当前节点。

（2）status：查询指定的节点的服务状态。

peer version 子命令用于查看当前 peer 命令的版本，等同于-v。

可以看到，我们通过 peer 命令可以使用到 Fabric 网络的大多数功能，也可以在步骤 r 的 bash 命令行中尝试执行这些命令并查看效果。在启动完 org1peer1 之后，我们会用同样的方式启动其他三个节点，因为节点 1 已经使用 Chaincode 与 Peer 节点通信的默认端口 7052，因此对其他三个节点使用的端口需要另外指定：

```
# 设置Chaincode的地址与Peer节点监听的地址
-e CORE_PEER_CHAINCODEADDRESS=$INTERNAL_IP:7053
-e CORE_PEER_CHAINCODELISTENADDRESS=$INTERNAL_IP:7053
```

在 4 个节点都启动后，在第 10 步将会通过 configtxgen 命令生成通道配置交易文件，以及两个通道的 AnchorUpdate 配置交易文件，然后通过 peer channel create 命令向 Orderer 节点提交创建通道请求；configtxgen 也是 Fabric 中的常用命令，除了生成创世区块及配置交易文件，还有许多其他用途，其参数及具体用法请参考 4.2.1 节。

这样对于生产过程中的任意区块文件和配置交易，我们都可以通过 peer 命令提取出来，然后通过 configtxgen 命令将这些文件解析出来并检查其中的内容，这在 Fabric 开发中是一种常见的开发调试手段。步骤 k ~ n 都是获取新通道的创世区块，然后将不同的节点加入通道中。步骤 o、p 则是将步骤 j，也就是通过在 createChannel 中生成的 AnchorUpdate 通道配置交易文件更新通道配置，设定两个组织的 AnchorPeer。在这些步骤中，我们需要注意 CORE_PEER_ADDRESS、CORE_PEER_LOCALMSPID 与 CORE_PEER_MSPCONFIGPATH 的变化，这三者的变化往往是同步的，CORE_PEER_ADDRESS 指向我们要操作的 Peer

节点，CORE_PEER_LOCALMSPID 是我们在提交提案时声明的 MSPID，CORE_PEER_MSPCONFIGPATH 则是我们声明的自己的身份。如果三者不一致，例如我们使用 Org2 的身份请求 Org1 的节点，或者使用 Org1 的身份请求 Org1 的节点，但 MSPID 是 Org2，则都会报错。这也让我们对 MSP 建立了一个初步的印象。

接下来通过步骤 q 安装 Chaincode。可以看到，通过 peer chaincode install 命令传入了 Chaincode ID 的三要素——Name、Version、Path；在步骤 r 中启动了一个 bash，可以输入命令操作 Chaincode。事实上，除了可以使用 peer chaincode 命令操作 Chaincode，还可以尝试 peer 命令的其他用法。

通过 docker ps 命令，我们可以看到 Chaincode 只有一个：nid1-jdoe-example02-v1；通过 docker inspect nid1-jdoe-example02-v1 命令，我们看到在这个容器启动命令中指定监听的是 7052 端口，也就是 org1peer1 的端口，那么为什么启动的是 org1peer1 的 Chaincode 呢？怎样启动其他节点的 Chaincode 呢？在一个节点的 Chaincode 上的交易可以在其他节点的 Chaincode 上查询到吗？希望大家能够用已经掌握的知识在我们提供的 Fabric_Demo 中动手尝试解决这些问题。

第4章 Fabric应用开发实践

4.1 Fabric SDK概述

之前的章节讲解了如何使用命令行部署和调用智能合约，本章将从应用开发的角度来讲解如何使用Fabric提供的开发工具包（Fabric SDK）来开发区块链应用。Fabric SDK采用了Google开源的gRPC远程通信框架。gRPC是一个通用的遵循HTTP/2协议的高性能开源RPC框架，主要面向移动应用的开发，基于Protocol Buffers序列化协议开发而成。Protocol Buffers支持很多语言，典型的如Go、Java、C#、JavaScript、Python和C++等。

Fabric SDK提供的API主要涵盖以下几方面。

（1）通道的生命周期管理。

（2）智能合约的生命周期管理。

（3）以编程方式调用、执行智能合约。

（4）各类查询接口，例如查询区块（Block）信息、交易（Transaction）信息。

（5）事件监听，例如监听交易事件、区块事件。

总体上，Fabric SDK（以Java版本为例）主要分为以下几个模块。

（1）Client模块：是Fabric SDK的主要入口模块，我们首先要创建一个Fabric Client

对象，并且通过这个 Client 对象连接到 Fabric 网络中，才能执行后续的一些操作，例如执行智能合约部署、调用及查询。

（2）Chains 模块：提供了通道的操作方法。在这里，一个通道代表一个业务网络中的一类业务，一个业务网络可以包含多个通道，通道中的多个 Peer 节点维护着包含链上事务、成员资格配置等在内的分类账本。

除此之外，还包括以下几个辅助模块。

（1）Peer 模块：提供了对 Fabric Peer 节点对象的封装。

（2）Orderer 模块：提供了对 Fabric Orderer 节点对象的封装。基于 Fabric 的区块链网络应该拥有一个或多个 Orderer 节点，应用程序可以选择某个特定的 Orderer 节点或者一组 Orderer 节点，或者为多个排序服务节点设立一个代理来传播交易请求。

（3）User 模块：提供了对 Fabric User 对象的封装。在通常情况下，User 也可以代表 Peer 节点的身份，但是在 SDK 的设计中并没有体现这一点，应用程序应该更多地关注网络管理问题。

（4）Member Service 模块：主要目标是从成员服务中获取用户的注册证书，也提供了对用户的注册、用户身份的注册及组织结构（从属关系）的管理功能。

这里首先详细讲解 Client 和 Chains 模块的用法，在第 6 章中将详细讲解 User 和 Member Service 模块的用法。

4.1.1 Client 模块

在现实世界中会存在多个彼此独立的区块链网络。每个区块链网络都包含 Peer 节点、Orderer 节点及 Fabric CA 节点，Client 模块负责与区块链网络中的 Peer 节点、Orderer 节点及 Fabric CA 节点通信。应用程序可以通过 SDK 与多个区块链网络通信，每个区块链网络都通过一个独立的 Client 对象的实例通信。每个 Client 对象的实例在初始化时，都需要包含一组可以信任的根证书、Orderer 节点的证书和 Orderer 节点的 IP 地址，以及它有权访问的 Peer 节点的证书和 IP 地址列表。

每个 Client 对象的实例都可以维护一个区块链网络中的多个通道。接下来，我们看看 Client 都提供了哪些 API。

（1）HFClient.newChannel：可通过以下两种方法创建通道实例。

- HFClient.newChannel(String name)：创建一个已经存在的通道对象的实例。
- HFClient.newChannel(String name, Orderer orderer, ChannelConfiguration channelConfiguration,byte[]... channelConfigurationSignatures)：创建一个新的通道并返回该通道对象的实例。该方法的第 2 个参数 orderer 是一个 Orderer 节点对象的实例，如果在一个区块链网络中存在多个 Orderer 节点，则指定其中一个 Orderer 节点对象的实例即可，其他 Orderer 节点会收到通道创建的配置区块，而配置区块是独立成块的；第 3 个参数 channelConfiguration 是通道的配置文件；第 4 个参数 channelConfigurationSignatures 是通道配置文件的签名。

（2）HFClient.getChannel(String name)：从应用程序的状态数据库中获取一个通道对象的实例，fabric-sdk-java 实现的通道对象的实例目前被保存在内存中；通道对象的实例是可以被序列化和反序列化的，应用程序也可以根据自己的需求将通道对象的实例由保存在内存中修改为保存在数据库中。

（3）HFClient.queryChannels(Peer peer)：查询指定的 Peer 节点加入的通道的名称。指定的 Peer 对象的实例必须是通道的一部分。

（4）HFClient.setCryptoSuite(CryptoSuite cryptoSuite)：设置一个实现了 CryptoSuite 模块接口的实例。

（5）HFClient.getCryptoSuite()：获取当前客户端的凭证套件。

（6）HFClient.setUserContext(User userContext)：设置当前客户端的用户身份。在区块链网络中发起交易及查询时都将用到这个用户的注册证书（Ecert）。

（7）HFClient.getUserContext()：获取当前客户端的用户身份。

Client 模块中的 API 及其说明如表 4-1 所示。

表 4-1

API	说　明
deSerializeChannel	反序列化通道实例
getChannelConfigurationSignature	获取通道配置文件的签名
getUpdateChannelConfigurationSignature	对通道的配置更新文件进行签名

续表

API	说　明
loadChannelFromConfig	从配置中加载通道实例
newEventHub	创建一个新的 Peer 事件节点，Fabric 从 1.1.0 版本开始，已经不再推荐使用该 API
newInstallProposalRequest	创建安装智能合约交易提案
newInstantiationProposalRequest	创建智能合约实例化交易提案
newOrderer	创建 Orderer 节点的终端实例
newPeer	创建 Peer 节点的终端实例
newQueryProposalRequest	创建智能合约查询提案
newTransactionProposalRequest	创建智能合约交易提案
newUpgradeProposalRequest	创建智能合约更新提案，智能合约在升级时会用到该提案
queryInstalledChaincodes	查询指定的 Peer 节点已经安装的智能合约
sendInstallProposal	向指定的 Peer 节点发送安装智能合约的交易提案。需要执行智能合约的节点都需要安装智能合约。在区块链网络运行一段时间后，新加入的节点如果要执行智能合约，则也需要安装智能合约

4.1.2　Chains 模块

我们可以将 Chains 模块理解为一个通道，通道由 Orderer 节点创建，用于隔离不同的通道之间的事务，通道在配置完 Peer 节点列表和 Orderer 节点列表后，必须被初始化才能使用，在初始化后会开始监听 Peer 节点的事件。接下来，我们看看 Chains 模块都提供了哪些方法。

（1）Channel.addPeer：增加一个 Peer 节点实例到通道实例中。该方法操作的是本地对象，并不会立即与 Peer 节点建立链接，在通道实例初始化时才会与 Peer 节点建立链接。它包含以下两种方法。

- Channel.addPeer(Peer peer)：添加一个 Peer 节点实例到通道实例中，当前的 Peer 节点拥有全部角色，包括背书节点（ENDORSING_PEER）、智能合约查询节点（CHAINCODE_QUERY）、账本查询节点（LEDGER_QUERY）及事件监听节点（EVENT_SOURCE）。
- Channel.addPeer(Peer peer,PeerOptions peerOptions)：添加一个 Peer 节点实例到通道实例中，并为当前的 Peer 节点指定角色。

（2）Channel.removePeer(Peer peer)：从当前的通道实例中删除一个 Peer 节点实例。该方法操作的是本地对象，当前的 Peer 节点仍然在由通道组成的业务网络中。

（3）Channel.getPeers：获取当前通道实例中的 Peer 节点实例，包含以下两种方法。

- Channel.getPeers()：获取当前通道实例中的 Peer 节点实例，不管 Peer 对象是什么角色。
- Channel.getPeers(EnumSet<PeerRole> roles)：只获取在当前通道实例中拥有指定角色的 Peer 节点实例。

（4）Channel. joinPeer：将 Peer 节点加入由通道组成的业务网络中并返回 Peer 节点实例。与 addPeer 方法不同，该方法操作的是远程对象，在将 Peer 节点加入由通道组成的业务网络中后，返回 Peer 节点实例，同时，Peer 节点开始同步新加入的通道的账本数据。它包含以下三种方法。

- Channel. joinPeer(Peer peer)：将 Peer 节点加入由通道组成的业务网络中并返回 Peer 节点实例，新添加的 Peer 节点实例拥有全部角色，同时从 Orderer 节点集合中随机选择一个 Orderer 节点来拉取通道的创世区块。
- Channel. joinPeer(Peer peer, PeerOptions peerOptions)：将 Peer 节点加入由通道组成的业务网络中，并返回 Peer 节点实例；将指定的角色赋予 Peer 节点对象，同时从 Orderer 节点集合中随机选择一个 Orderer 节点来拉取通道的创世区块。
- Channel. joinPeer(Orderer orderer, Peer peer, PeerOptions peerOptions)：将 Peer 节点加入由通道组成的业务网络中，并返回 Peer 节点实例；将指定的角色赋予 Peer 节点对象，同时向指定的一个 Orderer 节点发送请求来拉取通道的创世区块。

（5）Channel.addOrderer(Orderer orderer)：增加一个 Orderer 对象实例到通道实例中。该方法操作的是本地对象。

（6）Channel.getOrderers()：获取一个通道实例中的所有 Orderer 节点实例。

（7）Channel.initialize()：初始化一个通道实例，在初始化完成后，将当前的通道实例加入 Client 对象实例中。

（8）Channel.updateChannelConfiguration：更新通道配置文件，包含以下两种方法。

- Channel.updateChannelConfiguration(UpdateChannelConfigurationupdateChannelConfiguration, byte[]... signers)：从 Orderer 节点集合中随机选择一个 Orderer 节点来更新当前通道的配置。
- Channel.updateChannelConfiguration(UpdateChannelConfigurationupdateChannelConfiguration, Orderer orderer, byte[]... signers)：指定一个 Orderer 节点来更新当前通道的配置。

（9）Channel.queryBlockchainInfo：查询当前通道的区块信息，包含以下 4 种方法。

- Channel.queryBlockchainInfo()：向拥有账本查询（LEDGER_QUERY）角色的一个 Peer 节点发送请求来获取当前通道的区块信息，使用 Client 当前用户的身份。
- Channel.queryBlockchainInfo(User userContext)：向拥有账本查询（LEDGER_QUERY）角色的一个 Peer 节点发送请求来获取当前通道的区块信息，使用指定的用户身份。
- Channel.queryBlockchainInfo(Peer peer)：向指定的 Peer 节点发送请求来获取当前通道的区块信息，使用 Client 当前用户的身份。
- Channel. queryBlockchainInfo(Peer peer, User userContext)：向指定的 Peer 节点发送请求来获取当前通道的区块信息，使用指定的用户身份。

（10）Channel. queryBlockByNumber：根据区块编号查询账本区块，包含以下 6 种方法。

- Channel.queryBlockByNumber(long blockNumber)：向拥有账本查询（LEDGER_ QUERY）角色的一个 Peer 节点发送请求来获取账本区块，使用 Client 当前用户的身份。
- Channel.queryBlockByNumber(long blockNumber, User userContext)：向拥有账本查询（LEDGER_QUERY）角色的一个 Peer 节点发送请求来获取账本区块，使用指定的用户身份。
- Channel. queryBlockByNumber(Peer peer, long blockNumber)：向指定的 Peer 节点发送请求来获取账本区块，使用 Client 当前用户的身份。
- Channel. queryBlockByNumber(Peer peer, long blockNumber, User userContext)：向指定的 Peer 节点发送请求来获取账本模块，使用指定的用户身份。
- Channel. queryBlockByNumber(Collection<Peer> peers, long blockNumber)：向指定

的 Peer 节点集合中随机的一个 Peer 节点发送请求来获取账本模块，使用 Client 当前用户的身份。

- Channel.queryBlockByNumber(Collection<Peer> peers, long blockNumber, User userContext)：向指定的 Peer 节点集合中随机的一个 Peer 节点发送请求来获取账本模块，使用指定的用户身份。

（11）Channel.sendTransactionProposal：发送智能合约交易提案到当前通道拥有的所有背书节点，包含以下两种方法。

- Channel. sendTransactionProposal(TransactionProposalRequest transactionProposal Request)：发送交易提案到当前通道拥有的所有背书节点。
- Channel.sendTransactionProposal(TransactionProposalRequest transactionProposalRequest, Collection<Peer> peers)：发送交易提案到当前通道中指定的背书节点。

（12）Channel.sendTransaction：向当前通道实例中的一个 Orderer 节点发送交易，包含以下 5 种方法。

- Channel.sendTransaction(Collection<ProposalResponse> proposalResponses, User userContext)：向当前通道实例中的一个 Orderer 节点发送交易，使用指定的用户身份。
- Channel.sendTransaction(Collection<ProposalResponse> proposalResponses)：向当前通道的 Orderer 服务中的一个节点发送交易，使用 Client 当前用户的身份。
- Channel.sendTransaction(Collection<ProposalResponse> proposalResponses, Collection <Orderer> orderers)：向指定的一个 Orderer 节点发送交易，使用 Client 当前用户的身份。
- Channel.sendTransaction(Collection<ProposalResponse> proposalResponses, Collection <Orderer> orderers, User userContext)：向指定的一个 Orderer 节点发送交易，使用指定的用户身份。
- Channel.sendTransaction(Collection<ProposalResponse>proposalResponses, TransactionOptions transactionOptions)：使用自定义的交易选项向 Orderer 节点提交交易。通过该方法的第 2 个参数 TransactionOptions 可以设置以下自定义选项。
  - ✧ 设置交易用户的身份。
  - ✧ 设置 Orderer 节点集合。
  - ✧ 设置是否随机从 Orderer 节点集合中选择一个 Orderer 节点来发起交易。
  - ✧ 设置是否监听交易事件。

（13）Channel. queryByChaincode：查询智能合约，包含以下两种方法。

- queryByChaincode(QueryByChaincodeRequest queryByChaincodeRequest)：向拥有智能合约查询（CHAINCODE_QUERY）角色的 Peer 节点发送查询请求。
- queryByChaincode(QueryByChaincodeRequest queryByChaincodeRequest, Collection <Peer> peers)：向指定的 Peer 节点集合中的所有 Peer 节点发送智能合约查询请求。

（14）Chains 模块的其他 API 及其说明如表 4-2 所示。

表 4-2

API	说　明
getEventHubs	获取事件中心列表，从 fabric-sdk-java 1.1.0 版本开始已经不推荐使用
isInitialized	判断当前通道是否已经初始化
getName	获取当前通道的名称
getPeersOptions	获取指定的 Peer 节点的角色
addEventHub	添加一个事件中心，从 fabric-sdk-java 1.1.0 版本开始已经不推荐使用
isShutdown	判断通道是否已经关闭
sendInstantiationProposal	发送智能合约实例化请求交易提案
sendUpgradeProposal	发送升级智能合约交易提案
queryBlockByHash	根据哈希值查询区块信息
queryBlockByTransactionID	根据事务 ID 查询区块信息
queryInstantiatedChaincodes	查询已经实例化的智能合约
registerBlockListener	注册区块监听事件
unregisterBlockListener	取消注册区块监听事件
registerChaincodeEventListener	注册自定义智能合约事件
unregisterChaincodeEventListener	取消注册区块监听事件

续表

API	说　明
shutdown	关闭通道
serializeChannel	序列化通道
getChannelConfigurationBytes	获取当前通道的配置信息

4.2　通道配置

由 Fabric 构建的区块链网络，通过通道技术来实现账本数据的私密性，交易只在参与方之间的通道中传播，数据也只保存在参与方之间的分布式账本中。不在通道中的节点，无法看到通道中的任何信息。

4.2.1　使用 Configtxgen 工具生成通道配置

Configtxgen 工具是一段可以独立运行的程序，可以创建排序服务节点的创世区块，生成通道创世区块并更新通道的锚定节点配置，其中，锚定节点是在组织内部负责与组织外部通信的节点。我们可以通过以下两种方式获取 Configtxgen 工具。

（1）通过编译 Fabric 源码获取。可以在 fabric 目录下执行 make configtxgen，在执行成功后会在 fabric/build/bin 目录下生成 Configtxgen 工具。

（2）直接从 Docker 镜像 hyperledger/fabric-tools 中获取。运行以下命令新建一个窗口：

```
docker run -it --rm --name configtxgen bash
```

运行以下命令，将容器中的 Configtxgen 工具复制到当前目录下：

```
docker cp configtxgen:/usr/local/bin/configtxgen  ./
```

在使用 Configtxgen 工具时需要用到 congfigtx.yaml 配置文件。configtx.yaml 配置文件描述了区块链网络的组织信息，可以根据 configtx.yaml 生成 Orderer 创世区块及通道配置区块。关于 configtx.yaml 的文件结构，会在第 5 章中详细介绍。

我们可以通过运行 configtxgen help 命令获取 Configtxgen 工具的参数，如表 4-3 所示。

表 4-3

参　数	类　型	说　明
-asOrg	string	组织的名称
-channelID	string	通道的名称
-inspectBlock	string	以 JSON 格式显示排序服务创世区块中的内容
-inspectChannelCreateTx	string	以 JSON 格式显示通道创世区块中的内容
-outputAnchorPeersUpdate	string	创建锚定节点配置
-outputBlock	string	生成排序服务创世区块（配置区块）
-outputCreateChannelTx	string	生成通道创世区块（配置区块）
-printOrg	string	以 JSON 格式显示组织定义信息
-profile	string	指定 configtx.yaml 中 Profiles 的配置项
-version	string	显示版本信息

接下来通过 Configtxgen 工具生成通道配置区块：

```
  configtxgen -profile OneOrgChannel -outputCreateChannelTx channel1.tx -channelID
channel1
```

上面的命令根据 configtx.yaml 文件的 Profiles 配置项中的 OneOrgChannel，配置生成了一个名为 channel1 的通道配置区块。在创建通道时，会用到这个通道配置区块。

4.2.2　创建通道

在创建新通道时需要一个包含新链密钥、参与者信息、排序节点信息及新通道权限策略的创世区块，以及创建通道的配置文件（configtx.yaml）。通道配置区块的内容可以通过 Fabric 的 Configtxgen 工具生成。

创建通道的步骤如下。

（1）运行 Configtxgen 工具，生成通道配置区块文件，这里已经生成了一个 channel1.tx 文件。

（2）获取签名的通道配置文件，有以下两种方式。

方式一，使用生成的 channel1.tx，利用 Client 模块的 getChannelConfigurationSignature 方法进行签名。下面这行代码的 channelPath 参数为 channel1.tx 的文件路径：

```
ChannelConfiguration channelConfiguration = new ChannelConfiguration(new
File(channelPath));
client.getChannelConfigurationSignature(channelConfiguration, user);
```

方式二，构建自定义的通道配置文件，最简单的方式是使用 Configtxlator 工具将一个已存在的通道配置文件（channel1.tx）转换成 JSON 格式的文本。步骤如下。

首先，运行以下命令启动 Configtxlator 服务：

```
./configtxlator  start
```

然后，发送 channel1.tx 给 Configtxlator 服务并将其转换成 JSON 文本。JSON 文本也可以作为一个模板创建其他通道。需要注意的是，新创建通道的组织必须和系统通道属于同一个联盟。运行以下命令，将产生一个 channel1.json 文件：

```
curl -X POST --data-binary @channel1.tx
http://127.0.0.1:7059/protolator/decode/common.Envelope >channel1.json
```

其次，将 channel1.json 作为创建通道的模板，并根据实际需求修改相应的策略。

最后，通过 Configtxlator 工具将编辑后的 JSON 文本转换成通道配置定义文件：

```
curl -X POST --data-binary @channel1.json
http://127.0.0.1:7059/protolator/encode/common.Envelope > channel1.tx
```

（3）使用 SDK 对 channel1.tx 进行签名。

首先，构建通道配置对象：

```
ChannelConfiguration channelConfiguration = new ChannelConfiguration(new
File(channelPath));      // channelPath 为 channel1.tx 的文件路径
```

然后，使用 user 用户身份进行签名，这里的 user 用户一般为 Peer 管理员或组织管理员：

```
byte[] signature =client.getChannelConfigurationSignature(channelConfiguration,
user);
```

（4）通过 HFClient 创建通道：

```
Channel newChannel = client.newChannel(channelName, orderers.get(0),
```

```
channelConfiguration,signature);
```

（5）使用 SDK 将当前的 Peer 节点加入通道中：

```
channel.joinPeer(peer);
```

（6）完成通道的初始化：

```
channel.initialize();
```

4.2.3 加入通道

在 4.2.2 节中，细心的读者会发现，在通道创建完成后，Peer 节点已经加入通道中。本节把加入通道单独作为一个例子来讲解，因为在通道创建完成后，其他节点在加入通道时不再需要创建通道，并且在通道运行一段时间后，可能有新节点加入通道。

新节点在加入通道后，需要同步账本的所有数据。而账本是链式结构的，所以在加入通道时，需要先获取通道的创世区块，而不是最新的配置区块，与创建通道后立即加入通道不同。以下是通过命令行方式加入通道的例子：

```
peer channel fetch 0  -o ${ORDERER_URL} -c channel1;
peer channel join -b channel1_0.block
```

在上面的例子中拉取的是通道的创世区块，而不是最新的配置区块。

接下来用 fabric-sdk-java 演示新节点加入通道中的过程：

```
    // 获取通道实例
    Channel channel  = client.getChannel(channelName);
    // 将当前Peer节点加入通道中
channel.joinPeer(peer);
    // 完成初始化
    channel.initialize();
```

4.2.4 更新通道

Fabric 提供了 Configtxlator 工具，用于独立于 SDK 重新配置通道，旨在通过 API 与 SDK 进行交互，以协助完成配置更新操作。Configtxlator 工具本身并不生成配置，也不提交或者检索配置，更不会修改配置，而是将数据的 JSON 格式和二进制格式进行转换。我们在创建通道时，已经使用了 Configtxlator 工具生成通道的配置文件。与创建通道不同的

是，在更新通道时使用 SDK 修改了 JSON 文件，而不是手工修改 JSON 文件。

使用 Configtxlator 工具完成通道配置的步骤如下。

（1）使用 SDK 获取最新的配置，通过通道的 getChannelConfigurationBytes 方法获取当前通道的配置信息：

```
byte[] channelConfBytes = channel.getChannelConfigurationBytes();
```

（2）使用 Configtxlator 工具将配置信息转换成 JSON 格式：

```
HttpClient httpclient = HttpClients.createDefault();
HttpPost httppost = new HttpPost("http://127.0.0.1:7059/protolator/decode/
common.Config");
httppost.setEntity(new ByteArrayEntity(channelConfBytes));
HttpResponse response = httpclient.execute(httppost);
int statuscode = response.getStatusLine().getStatusCode();
assertEquals(200, statuscode);
String responseAsString = EntityUtils.toString(response.getEntity());
```

（3）通过应用程序编辑配置，这里将区块生成的超时时间由两秒修改为五秒，即如果有交易产生，那么最多 5 秒会产生一个区块：

```
String ORIGINAL_BATCH_TIMEOUT = "\"timeout\": \"2s\"";
String UPDATED_BATCH_TIMEOUT = "\"timeout\": \"5s\"";
String updateString = responseAsString.replace(ORIGINAL_BATCH_TIMEOUT,
UPDATED_BATCH_TIMEOUT);
httppost = new HttpPost("http://127.0.0.1:7059/protolator/encode/common.Config");
httppost.setEntity(new StringEntity(updateString));
response = httpclient.execute(httppost);
statuscode = response.getStatusLine().getStatusCode();
byte[] newConfigBytes = EntityUtils.toByteArray(response.getEntity());
```

（4）使用 Configtxlator 工具计算配置更新的内容：

```
httppost = new HttpPost("http://127.0.0.1:7059/configtxlator/compute/update-from-
configs");
HttpEntity multipartEntity = MultipartEntityBuilder.create();
.setMode(HttpMultipartMode.BROWSER_COMPATIBLE)
.addBinaryBody("original", channelConfigurationBytes,
ContentType.APPLICATION_OCTET_STREAM, "originalFakeFilename")
.addBinaryBody("updated", newConfigBytes, ContentType.APPLICATION_OCTET_STREAM,
"updatedFakeFilename")
.addBinaryBody("channel", channel.getName().getBytes()).build();
```

```
    httppost.setEntity(multipartEntity);
    response = httpclient.execute(httppost);
    statuscode = response.getStatusLine().getStatusCode();
    assertEquals(200, statuscode);
    byte[] updateBytes = EntityUtils.toByteArray(response.getEntity());
```

（5）使用 SDK 对配置更新的内容进行签名，并将其提交给 Orderer 节点：

```
    // 需要使用orderer admin用户进行签名
    client.setUserContext(ordererAdmin);
    UpdateChannelConfiguration updateChannelConfiguration = new
UpdateChannelConfiguration(updateBytes);
    // 调用Client的updateChannelConfiguration的方法更新通道配置
    channel.updateChannelConfiguration(updateChannelConfiguration,
client.getUpdateChannelConfigurationSignature(updateChannelConfiguration,
ordererAdmin));
    // 在更新完成后，需要将当前Client用户的身份切换回当前Peer管理员的身份
    client.setUserContext(peerAdmin);
```

在上面的例子中更新了生成通道区块的超时时间。也可以尝试修改其他配置项，研究是否可以修改所有的配置项，比如是否可以在创建后修改共识类型。现在已经完成了通道配置的更新工作，涉及通过手工修改通道配置和通过应用程序修改通道配置两种方式。

4.3 智能合约管理

作为区块链 2.0 的重要特性，智能合约大大扩展了区块链的应用范围，从简单地记账升级成了根据输入触发一系列操作的行为。以太坊为了能够在公开环境下创造一个安全、隔离的运行环境，设计了 EVM（Ethereum Virtual Mathine），在其上运行着以 Solidity 为代表的智能合约语言。Fabric 面向私有或者半公开的环境，并没有选择自己重新开发智能合约开发语言及虚拟机，而是使用了很成熟的 Docker 技术与现有的语言，通过容器、API、网络结构等方式，一方面保护智能合约的运行环境，另一方面限制智能合约的行为。

保护智能合约的运行环境，主要是为了防止智能合约的执行因为外界因素的影响而产生错误的结果。而对于为什么限制智能合约的行为，则不容易让人理解，通常我们可以从以下三个角度来考虑其中的原因。

（1）智能合约的权限：首先，在调用智能合约时并不代表当前交易已经结束，智能合约本身并不能确定交易的结果，因此，智能合约的运行结果不能被直接写入账本中；其次，

智能合约是运行在特定的通道中的，在同一 Peer 节点的不同通道中的智能合约需要禁止跨通道的数据访问，因为这样也会打破通道的数据隔离原则。

（2）智能合约的安全性：如果不考虑 Fabric 自身可能存在的缺陷和运维方面的漏洞，则智能合约是最有可能发生数据泄露的，除了可以将 Peer 节点与智能合约之间的通信通过 TLS 加密防止窃听，通过 Docker 环境的网络设置也可以防止智能合约泄露数据。

（3）交易的原子性：智能合约在 Fabric 的交易中是非常重要的一环，交易本身是需要保证原子性的，也就是说，如果交易失败，那么在该过程中需要还原到交易之前的状态。如果在智能合约运行的过程中再次发起交易或者调用其他智能合约，则有可能打破这种交易的原子性状态。因此，Fabric 限制只能修改同一个通道内的账本状态，其他通道的账本状态只能通过智能合约查询（注意是否有读取权限），不能修改。虽然没有硬性限制由智能合约通过集成 SDK 的方式发起交易，但强烈建议不要进行这样的尝试，因为自定义的智能合约在调用阶段不能确认交易是否成功。如果一定要在交易成功后发起另一个交易，则可以考虑通过监听交易成功事件的方式实现，或者自定义 VSCC，在确认交易成功后再发起交易。

4.3.1　开发智能合约

Fabric 中的智能合约也被称为 Chaincode。Chaincode 支持 Go、Java、Node.js 等多种语言开发，下面以 Go 语言版本的代码为例讲解 Chaincode 的开发。在开发 Chaincode 时，首先需要关注 core/chaincode/shim/interface_stable.go 文件，在这个文件中提供了 Chaincode 框架及其可用的 API。

Chaincode 框架提供的 API 及其说明如表 4-4 所示。

表 4-4

API	说　明
Init(stub ChaincodeStubInterface) pb.Response	在 Chaincode 初始化时调用，可以在其内部对一些数据进行初始化
Invoke(stub ChaincodeStubInterface) pb.Response	用于实现 invoke 与 query 功能，系统在调用 Chaincode 时，将参数作为字符串传入 Invoke 接口

在 Init 与 Invoke 接口中，传入的参数类型都是 ChaincodeStubInterface，这个类不但提供了 Chaincode 接口必需的参数，还提供了 Chaincode 运行的主要 API，如下所述。

（1）GetArgs() [][]byte：若在调用 Chaincode 时参数以字节数组形式传递，则 GetArgs 将参数以字节数组形式提供给对应的接口，在设计 Chaincode 时需要和调用方确认参数的传递形式。

（2）GetStringArgs() []string：若在调用 Chaincode 时参数以字符串形式传递，则 GetArgs 将参数以字符串数组形式提供给对应的接口。

（3）GetFunctionAndParameters() (string, []string)：与 GetStringArgs 类似，区别在于第 1 个返回值是发起调用的函数名称，第 2 个返回值是字符串数组形式的参数。

（4）GetArgsSlice() ([]byte, error)：将参数以一个字节属性的形式提供。

（5）GetTxID() string GetTxID：返回交易提案的 TxID。

（6）GetChannelID() string：获取调用时的通道 ID。

（7）InvokeChaincode(chaincodeName string, args [][]byte, channel string) pb.Response：用于调用其他 Chaincode 的 API，Chaincode 在调用其他 Chaincode 时可能会破坏交易的原子性，因此对这个接口的处理分为以下两种情况。

- 被调用的 Chaincode 与发起调用的 Chaincode 处于同一通道内，被调用的 Chaincode 产生的 RWSet 将会被添加到交易中。
- 被调用的 Chaincode 与发起调用的 Chaincode 处于不同的通道内，被调用的 Chaincode 产生的 RWSet 将不会被写入交易内，只会返回调用响应；也就是说跨通道的 Chaincode 调用只能执行查询功能，而不能产生交易或者修改状态。如果 chaincodeName 为空，则默认为当前通道。

（8）GetState(key string) ([]byte, error)：获取指定的 Key 在账本中的值，需要注意的是，GetState 不会读取未提交到账本中的值，因此有可能产生双花攻击，如何防范双花攻击，将会在第 5 章讨论。如果 Key 不存在，则会返回(nil, nil)。

（9）PutState(key string, value []byte) error：将指定的 Key 与 Value 放入交易提案的写集合（Write Set）中，写集合中的值需要在交易通过验证并且成功提交后才写入账本。Key 不能为空或者以空字符（0x00）为开始，避免与复合键（Composite Key）的前缀冲突。

（10）DelState(key string) error：用于将删除指定 Key 的请求放入提案的写集合中，在提案验证通过并成功提交后会从账本中删除该记录。

（11）GetStateByRange(startKey, endKey string) (StateQueryIteratorInterface, error)：返

回某个范围内的所有 Key，结果集以词法顺序返回，例如，GetStateByRange("a", "b")会返回 a 与 b 之间的所有 Key 如 a1、aaa、abc 等，但不包含 b。startKey 与 endKey 可以被设置为空，这意味着不限制范围的开始（startkey 为空）或者结束（startkey 为空）。该查询在验证阶段会重新执行，以确认结果集在交易背书后仍未改变（没有幻读）。

（12）CreateCompositeKey(objectType string, attributes []string) (string, error)：之前提到的查询都基于完整的 Key，如果希望查询特定属性的对象集合，则需要使用复合键（Composite Key）。CreateCompositeKey 可以将索引名（objectType）与属性列表（attributes）连接起来作为 Key 通过 PutState 放入账本中，连接格式为："\x00"+objectType+"\x00"+attributes[1]+"\x00"+attributes[2]+"\x00"...，是一个通过“\x00”分隔的 utf-8 字符串，因此在填入属性时需要确认索引名与属性值是否避开了 U+0000 与 U+10FFFF。

（13）SplitCompositeKey(compositeKey string) (string, []string, error)：是 CreateCompositeKey 的逆操作，会将复合键还原为索引名与属性列表。

（14）GetStateByPartialCompositeKey(objectType string, keys []string) (StateQueryIteratorInterface, error)：用于根据部分属性查询账本内对应的复合键，关于索引名与属性值的取值范围请参考 CreateCompositeKey 的说明。查询结果是一个 StateQueryIteratorInterface 迭代器，通过 Next 方法可以遍历结果集，在退出时需要调用 Close 方法。该方法查询的对象同样会在交易验证阶段进行幻读验证。

（15）GetQueryResult(query string) (StateQueryIteratorInterface, error)：用于构建一个针对状态数据库的富查询，在查询条件并未构建在复合键中且指定的状态数据库支持富查询时可以使用该功能。例如 CouchDB 支持富查询，LevelDB 则不支持富查询。查询的语法依赖于状态数据库的本地实现。需要注意的是，GetQueryResult 在当前版本（Fabric V1.1）中没有把结果集放入 ReadSet 中，也就是说 GetQueryResult 暂时不支持幻读验证，它的结果集只能用于只读结果查询。

（16）GetHistoryForKey(key string) (HistoryQueryIteratorInterface, error)：返回指定 Key 的所有历史记录。获取历史记录的前提是 Peer 节点配置中的 core.ledger.history.enableHistoryDatabase 为 true，会以{"TxId":...,"Values":...," Timestamp" :...," IsDelete":...}的格式返回，Timestamp 是客户端在交易提案中写入的时间。GetHistoryForKey 因为会返回所有历史记录，所以结果集不能作为幻读验证的依据，因此 GetHistoryForKey 只能用于只读结果查询。

（17）GetCreator() ([]byte, error)：用于提取签名交易提案（SignedProposal）中的签名

信息（SignatureHeader.Creator），也就是提交者的身份。

（18）GetTransient() (map[string][]byte, error)：用于提取交易提案中 ChaincodeProposalPayload.Transient 字段的值。Transient 字段通常用于保存有应用程序级保密性的数据（例如密码材料），并且最终不会存入账本。

（19）GetBinding() ([]byte, error)：返回由 protos/utils/proputils.go 中 ComputeProposalBinding 函数根据提案的 Nonce、Creator 与 Epoch 生成的一个哈希值，该值主要用于防止重放攻击。

（20）GetDecorations() map[string][]byte：返回由 core/handlers/decorations 中的装饰器生成的数据。

（21）GetSignedProposal() (*pb.SignedProposal, error)：返回 SignedProposal 对象，包含整个交易数据。

（22）GetTxTimestamp() (*timestamp.Timestamp, error)：返回交易提案生成时的时间戳。

（23）SetEvent(name string, payload []byte) error：将指定的事件名称（name）与内容（Payload）设置到交易提案中，在交易提案进行验证后会发送事件。

可以看到，Chaincode 的基础开发其实非常简单，通过实现 Init 与 Invoke 方法，即可实现 Chaincode 的基本功能。在我们之前使用的 example/chaincode/go/example02 中就实现了一个简单的账本功能，在 example/chaincode/go/marbles02 中使用了账本查询接口的高级功能，在 example/chaincode/go/example05 中使用了 InvokeChaincode 调用其他 Chaincode，等等。需要强调的是，到 Fabric 1.1 版本为止，不要将 GetQueryResult 的结果集作为修改账本的依据，因为查询的结果集不会被放在 ReadSet 中，无法进行幻读验证。所谓幻读，是指有可能交易成功的交易在途时，Chaincode 读取了还未更新的账本，又生成了一个新的交易，这时如果 Committer 无法对读取的第 2 条数据的版本进行验证，则会发生双花（*N*-Spending）现象。在第 5 章中，我们会讨论双花现象的另一种情况与防范方法。Fabric 作为数据库，事务隔离级别非常有限，对于这一点需要特别注意。

4.3.2 安装智能合约

我们可以将安装智能合约理解为将智能合约的代码上传到 Peer 节点。接下来以 Fabric 官网提供的以 Go 语言实现的 example02 智能合约为例，演示智能合约的安装过程。

关键代码如下：

```
        final ChaincodeID chaincodeID =
ChaincodeID.newBuilder().setName(chaincodeName)
                .setVersion(chaincodeVersion)
                .setPath(chaincodePath).build();
          // 构建InstallProposalRequest对象
          InstallProposalRequest installProposalRequest =
client.newInstallProposalRequest();
          installProposalRequest.setChaincodeID(chaincodeID);
          // 获取智能合约源码的两种方式
          if (true) {
             installProposalRequest.setChaincodeSourceLocation(new
File(chainRootPath));
          } else {
             installProposalRequest.setChaincodeInputStream(Util.
generateTarGzInputStream(
                   (Paths.get("src"+chaincodePath).toFile()),
                   Paths.get("src"+chaincodePath).toString()));
          }
          // 设置智能合约的版本号
          installProposalRequest.setChaincodeVersion(chaincodeVersion);
          // 设置在哪些节点安装智能合约，这里选择所有的背书节点
          EnumSet<Peer.PeerRole> roles =
EnumSet.complementOf(EnumSet.of(Peer.PeerRole.ENDORSING_PEER));
          Collection<Peer> peersFromOrg = channel.getPeers(roles);
          Collection<ProposalResponse> successful = new LinkedList<>();
          Collection<ProposalResponse> failed = new LinkedList<>();
          // 通过Client向背书节点发起安装智能合约的交易提案
          Collection<ProposalResponse> responses =
client.sendInstallProposal(installProposalRequest, peersFromOrg);
          // 省略校验背书节点响应结果的过程
```

4.3.3　实例化智能合约

在需要背书的节点安装智能合约后，智能合约还没有运行，我们需要对智能合约进行实例化，会与 Orderer（排序服务）节点进行交互，生成配置交易区块并同步到区块链网络中。

实例化智能合约的步骤如下。

（1）构建 InstantiateProposalRequest 对象。

（2）提交实例化智能合约提案给背书节点，进行交易验证。

（3）校验背书节点的响应结果。如果都验证成功，则继续下一步；如果有校验失败的背书节点，则结束流程。

（4）将实例化智能合约的响应结果发送给排序服务，生成配置区块。

关键代码如下：

```
        // 实例化智能合约
        InstantiateProposalRequest instantiateProposalRequest =
client.newInstantiationProposalRequest();
        instantiateProposalRequest.setChaincodeID(chaincodeID);
        // 设置在实例化时执行的智能合约的初始化方法
        instantiateProposalRequest.setFcn("init");
        // 设置在执行智能合约初始化方法时传递的参数
        instantiateProposalRequest.setArgs(new String[] {"a", "100", "b", "100"});
        // 指定背书策略
        ChaincodeEndorsementPolicy chaincodeEndorsementPolicy = new
ChaincodeEndorsementPolicy();
        chaincodeEndorsementPolicy.fromYamlFile(new File(endorsementpolicyFile));
        instantiateProposalRequest.setChaincodeEndorsementPolicy
(chaincodeEndorsementPolicy);
        Map<String, byte[]> tm = new HashMap<>();
        tm.put("HyperLedgerFabric",
"InstantiateProposalRequest:JavaSDK".getBytes(UTF_8));
        tm.put("method", "InstantiateProposalRequest".getBytes(UTF_8));
        instantiateProposalRequest.setTransientMap(tm);
        // 选择背书节点
        EnumSet<Peer.PeerRole> roles =
EnumSet.complementOf(EnumSet.of(Peer.PeerRole.ENDORSING_PEER));
        responses = channel.sendInstantiationProposal(instantiateProposalRequest,
channel.getPeers(roles));
        // 省略对响应结果进行校验的过程
        channel.sendTransaction(responses, orderers)
              .thenApply(transactionEvent -> {
                 assertTrue(transactionEvent.isValid());
                 return null;
              }).get(3, TimeUnit.SECONDS);
```

4.3.4　调用智能合约

智能合约在实例化后，会将配置区块广播到区块链网络中，但是只能在安装了智能合约的网络节点上调用，所以在区块链网络中所有需要参与背书的节点都需要安装智能合约。之所以放在这里说明这一点，是因为可以在实例化前安装智能合约，也可以在实例化后在其他节点上安装智能合约。

调用智能合约的步骤如下。

（1）构建 TransactionProposalRequest 对象。

（2）提交智能合约交易提案给背书节点，进行交易验证。

（3）校验背书节点的响应结果。如果都校验成功，则继续下一步；如果有背书节点校验失败，则结束流程。

（4）将智能合约交易提案发送给排序服务，通过排序服务生成交易区块。

关键代码如下：

```
        // 构建TransactionProposalRequest对象
        TransactionProposalRequest transactionProposalRequest =
client.newTransactionProposalRequest();
        transactionProposalRequest.setChaincodeID(chaincodeID);
        transactionProposalRequest.setFcn("invoke");
        transactionProposalRequest.setArgs(new String[] { "a", "b", "100"});
        // 提交智能合约交易提案给背书节点，进行交易验证
        Collection<ProposalResponse> transactionPropResp =
channel.sendTransactionProposal(transactionProposalRequest, channel.getPeers());
        // 省略对响应结果进行校验的过程
        // 向orderer节点发送交易
        BlockEvent.TransactionEvent transactionEvent =
channel.sendTransaction(successful);
        assertTrue(transactionEvent.isValid());
        System.out.println(format("Finished transaction with transaction id %s",
transactionEvent.getTransactionID()));
```

通过以下日志，可以看到交易已经完成：

```
Creating install proposal
sending transactionProposal to all peers with arguments: move(a,b,100)
Successful transaction proposal response Txid:
```

```
cdf17480467767dcae307f2f73e4744a85f5c1e8daa2f8437c92198525685b2 from peer testpeer
   Received 1 transaction proposal responses. Successful+verified: 1. Failed: 0
   Successful received transaction proposal responses.
   Sending chaincode transaction(move a,b 100) to orderer.
   Finished transaction with transaction id
cdf17480467767dcae307f2f73e4744a85f5c1e8daa2f8437c92198525685b2
```

4.3.5 查询智能合约

在完成智能合约的调用后，需要了解如何查询智能合约。在查询智能合约时需要经历以下步骤。

（1）构建 QueryByChaincodeRequest 对象。

（2）将查询请求发送到 Peer 节点进行查询。

可以发现，查询智能合约只在背书节点进行，也不会生成交易区块。

关键代码如下：

```
   final ChaincodeID chaincodeID = ChaincodeID.newBuilder().setName(CHAIN_CODE_NAME)
               .setVersion(CHAIN_CODE_VERSION)
               .setPath(CHAIN_CODE_PATH).build();
   // 构建QueryByChaincodeRequest对象
   QueryByChaincodeRequest queryByChaincodeRequest =
client.newQueryProposalRequest();
   queryByChaincodeRequest.setArgs(new String[] { "b"});
   // 设置调用智能合约中的query函数
   queryByChaincodeRequest.setFcn("query");
   queryByChaincodeRequest.setChaincodeID(chaincodeID);
   // 将交易发送到Peer节点进行查询
   Collection<ProposalResponse> queryProposals
   = channel.queryByChaincode(queryByChaincodeRequest, channel.getPeers());
   // 在queryProposals中保存了Peer节点返回的查询结果集。应用程序可以根据自己的业务需求，执行相应
的业务处理流程
```

4.3.6 升级智能合约

在升级智能合约时，需要先将新版本的智能合约安装到背书节点，之后还需要执行升级操作以使新版本的智能合约生效，升级操作的具体步骤如下。

（1）构建 UpgradeProposalRequest 对象。

（2）指定智能合约的背书策略。

（3）提交智能合约交易提案给背书节点进行交易验证。

（4）校验背书节点的响应结果。如果都验证成功，则继续下一步；如果有校验失败的背书节点，则结束流程。

（5）在将背书节点的响应结果发送给排序服务生成区块后，广播给区块链网络中的节点。

代码如下：

```
   final ChaincodeID chaincodeID = ChaincodeID.newBuilder().setName(CHAIN_CODE_NAME)
                .setVersion(CHAIN_CODE_VERSION)     // 为新的版本号
                .setPath(CHAIN_CODE_PATH).build();
          // 构建UpgradeProposalRequest对象
           UpgradeProposalRequest upgradeProposalRequest =
client.newUpgradeProposalRequest();
           upgradeProposalRequest.setChaincodeID(chaincodeID);
           upgradeProposalRequest.setFcn("init");
           upgradeProposalRequest.setArgs(new String[] {});
           // 指定背书策略
           ChaincodeEndorsementPolicy chaincodeEndorsementPolicy;
           chaincodeEndorsementPolicy = new ChaincodeEndorsementPolicy();
           chaincodeEndorsementPolicy.fromYamlFile(new File(TEST_FIXTURES_PATH +
"/sdkintegration/chaincodeendorsementpolicy.yaml"));
           upgradeProposalRequest.setChaincodeEndorsementPolicy
(chaincodeEndorsementPolicy);
   // 提交智能合约交易提案给背书节点，进行交易验证
   responses = channel.sendUpgradeProposal(upgradeProposalRequest);
   // 省略背书节点的响应过程。假设全部验证通过，则将背书节点的响应结果发送给排序服务
   channel.sendTransaction(responses,
sampleOrg.getPeerAdmin()).thenApply(transactionEvent -> {
              assertTrue(transactionEvent.isValid());
              return null;});
```

4.4 监听事件

在区块链网络中，由于参与方的角色或身份不同，需要监听的事件类型也会有所不同，可以从以下两个维度进行划分。

（1）第 1 个维度，根据事件服务类型进行划分。根据应用程序是否关心交易的内容，可以将事件服务类型进一步划分为以下两种。

- Deliver 服务：指发送已经提交到 Peer 账本的整个 Block。如果注册 Chaincode 事件，则可以在 ChaincodeActionPayload 中找到这些事件。
- Deliverfiltered 服务：指发送已经提交到 Peer 账本的 Block，但是不包含交易的内容。该服务可以使 Client 只接收与其交易及交易状态有关的信息。

（2）第 2 个维度，根据交易事件类型进行划分。在 Fabric 实现的区块链网络中会触发以下两种类型的交易事件。

- 连接到 Orderer 服务的客户端，在交易提交到 Orderer 节点后，会触发一个“事务已提交”或者“错误”的事件。
- 区块在被成功写入 Peer 账本后，会拥有 EVENT_SOURCE 角色，同时，支持 BLOCK、CHAINCODE 或 TRANSACTION 事件的 Peer 节点客户端会收到事务已完成或者错误的事件。

可将以上两种类型的交易事件进一步划分为以下四种。

- 已提交事件：适合只关心交易是否已经提交，并不关心交易是否成功或完成的场景。
- 监听交易事件：属于一次性事件，在交易完成后，事件自动取消。
- 监听智能合约事件：在一个通道中可以包含多个智能合约。应用程序可能并不需要监听所有智能合约的交易事件，这时可以监听某个或某几个智能合约的交易事件。
- 监听区块事件：一个区块在被提交到 Peer 账本后，会触发区块事件，在一个通道中可能包含多个智能合约，这也意味着在一个区块中可能会包含多个智能合约的交易事件。如果想监听一个通道中的所有智能合约（用户编写的智能合约及 LSCC）的区块生成时间，则可以监听区块事件。

4.4.1　事件服务类型

前面在介绍 Fabric SDK Client 模块和 Chains 模块的 API 时，曾提到 Fabric 从 1.1.0 版本开始，就已经不推荐使用 new EventHub 注册事件服务了，可以在基于通道的 PeerOptions 向通道中添加 Peer 节点时设置注册的服务类型，通道在初始化时完成注册。fabric-sdk-java 默认注册的是 Deliver 服务，当然，也可以显式注册 Deliver 服务。

以下为注册服务的示例代码：

```
Channel.PeerOptions peerOptions = Channel.PeerOptions.createPeerOptions();
// 设置注册Deliverfiltered服务类型
peerOptions.registerEventsForFilteredBlocks();
// 设置注册Deliver服务类型
peerOptions. registerEventsForBlocks();
// 将Peer节点及其服务类型添加到通道中
channel.addPeer(peer,peerOptions);
// 通道在初始化时完成对服务的注册
channel.initialize();
```

4.4.2　监听交易事件

在发送交易时，如果在发送交易的通道上注册了事件中心，并且在通道上存在 EVENT_SOURCE 角色的 Peer 节点，或者在发送交易时指定了监听事件，则 fabric-sdk-java 会自动为我们创建交易事件监听器，在交易完成后，交易事件监听器会自动取消注册。下面通过源码分析自动创建交易事件监听器的过程：

```
public CompletableFuture<TransactionEvent> sendTransaction(
Collection<ProposalResponse> proposalResponses,Channel.TransactionOptions
transactionOptions) {
    // 忽略无关代码
Channel.NOfEvents nOfEvents = transactionOptions.nOfEvents;
// 如果没有设置，则自动创建nOfEvents
        if (nOfEvents == null) {
      nOfEvents = Channel.NOfEvents.createNofEvents();
                Collection<Peer> eventingPeers = this.getEventingPeers();
                boolean anyAdded = false;
    // 如果当前通道存在EVENT_SOURCE角色的Peer节点，则自动注册交易事件监听器
                if (!eventingPeers.isEmpty()) {
                   anyAdded = true;
```

```
                    nOfEvents.addPeers(eventingPeers);
                }
        // 如果当前通道存在事件中心，则自动注册交易事件监听器
                Collection<EventHub> eventHubs = this.getEventHubs();
                if (!eventHubs.isEmpty()) {
                    anyAdded = true;
                    nOfEvents.addEventHubs(this.getEventHubs());
                }
        // 不需要交易事件监听器
                if (!anyAdded) {
                    nOfEvents = Channel.NOfEvents.nofNoEvents;
                }
            } else if (nOfEvents != NOfEvents.nofNoEvents) {
    // 如果存在nOfEvents，并且需要交易事件监听器，则进行合法性检查
                // 忽略合法性的代码检查
        }
    // 判断是否需要注册交易事件监听器
    final boolean replyonly = nOfEvents == NOfEvents.nofNoEvents
    || (getEventHubs().isEmpty() && getEventingPeers().isEmpty());

        CompletableFuture<TransactionEvent> sret;
        // 如果不需要注册交易事件监听器
        if (replyonly) {
            logger.debug(format("Completing transaction id %s immediately no event
hubs or peer eventing services found in channel %s.", proposalTransactionID, name));
            sret = new CompletableFuture<>();
        } else { // 注册交易事件监听器
            sret = registerTxListener(proposalTransactionID, nOfEvents,
transactionOptions.failFast);
        }
       // 忽略无关代码
    resp = orderer.sendTransaction(transactionEnvelope);
            lException = null; // no longer last exception .. maybe just failed.
    // 响应成功，这里只代表事务已成功提交
            if (resp.getStatus() == Status.SUCCESS) {
                success = true;
                break;
            } else {
                logger.warn(format("Channel %s %s failed. Status returned %s",
name, orderer, getRespData(resp)));
            }
```

```
        // 忽略无关代码
    if (success) {
                    logger.debug(format("Channel %s successful sent to Orderer
transaction id: %s", name, proposalTransactionID));
                    if (replyonly) {
    // 如果没有注册交易事件监听器，则直接回复交易提交成功
                        sret.complete(null);
                    }
                    return sret;
                }
```

如果需要验证提交的交易是否为有效交易，或者需要根据交易结果执行额外的操作，则可以通过注册交易事件监听器进行。

4.4.3　已提交事件

在 4.4.2 节，我们通过分析源码可以发现，如果我们不关心提交的交易是否为有效交易，只关心交易是否已经提交成功，则可以通过 TransactionOptions.nOfEvents (NOfEvents.nofNoEvents)设置不监听交易事件。

4.4.4　监听区块事件

在一个区块被成功写入账本后，会产生区块事件，并广播给已经注册了区块事件的应用程序客户端。在通道上注册区块监听事件的示例代码如下：

```
             channel.registerBlockListener(event->{
            // 一个区块可以包含多笔交易，所以在这里是一个集合
            event.getTransactionEvents().forEach(f->{
            // 获取交易ID
                out("transactionid is "+f.getTransactionID());
                // 获取交易结果，其他值可参见源码FabricTransaction类中的常量值
                out(" validationCode is " + f.getValidationCode());
                // 获取交易类型
                out(" transaction type is " + f.getType().name());
                f.getTransactionActionInfos().forEach(actionInfo->{
                    actionInfo.getTxReadWriteSet().getNsRwsetInfos().forEach
(rwSetInfo->{
                        try {
                            // 查询交易读写集合中的写集合
```

```
rwSetInfo.getRwset().getWritesList().stream().forEach(write->{
                            out("write key is " + write.getKey());
                            out("write value is " + write.getValue().toStringUtf8());
                        });
                    } catch (InvalidProtocolBufferException e) {
                        e.printStackTrace();
                    }
                });
            });
        });
    });
```

如图 4-1 所示是 Commiter 节点在接收到区块并经过 VSCC、MVCC 等验证后，写入本地账本的过程，可以看到，在更新完状态数据库后产生了区块事件。客户端发出的交易可能不会马上生成区块，而是在触发区块生成条件时，Orderer 节点才会将一组交易打包生成区块，并广播到区块链网络中。这也意味着客户端在提交交易后，可能并不会直接触发区块事件。

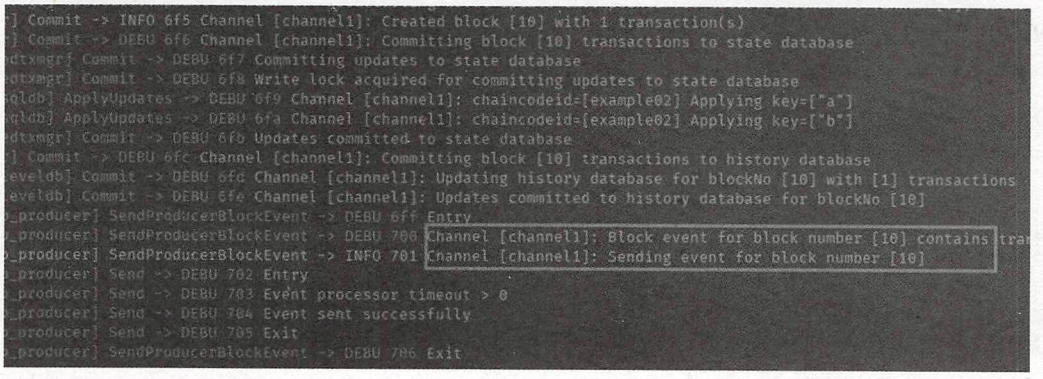


图4-1

4.4.5 智能合约事件

在有新的区块成功写入账本后会产生区块事件，在一个区块内可能包含多个智能合约产生的交易。如果应用程序只注册了区块事件，则在新区块产生后，应用程序会接收到区块生成事件；应用程序如果只关心某个智能合约的事件，并不想接收到其他智能合约的信息，就可以注册智能合约事件。

步骤如下。

（1）编写智能合约，设置响应事件及响应参数：

```
stub.SetEvent("example2_event_query", []byte(tosend))
```

（2）向通道注册 Chaincode 监听器：

```
channel.registerChaincodeEventListener(Pattern.compile(".*"),
            Pattern.compile(Pattern.quote("example2_event_query")),
            (handle, blockEvent, chaincodeEvent) -> {
      });
```

第5章
深入研究 Fabric 网络

在之前的章节中提到了，区块链是由区块组成的一个链条，每个区块都包含了当前的交易数据与上一个区块的哈希值，如果我们一路向上溯源，就会发现一个问题：创世区块是怎样的呢？如果没有上一个区块，那么它的结构是怎样的呢？

带着这个问题，我们回头看看比特币，在格林尼治标准时间 2009 年 1 月 3 日，中本聪生成了比特币的创世区块，在创世区块中有一句话：

```
The Times 03/Jan/2009 Chancellor on brink of second bailout for banks
```

这句话指的是《泰晤士报》2009 年 1 月 3 日的头条：英国财政大臣达林被迫考虑第二次出手缓解银行危机，如图 5-1 所示。

有人认为这是对金融危机下脆弱的银行系统的嘲讽，也有人认为这只是表示创世区块的创始时间晚于报纸的出版时间。无论如何，这值得区块链研究者思考，而这张报纸的头条也随着比特币的爆发而载入史册。

言归正传，Fabric 的设计者自然不会再去选择历史性的内容作为创世区块的内容，而会选择一些更为实用的内容。那么，一个刚刚启动的什么内容都没有的网络，什么内容才是最实用的呢？


THE TIMES

Eat Out from £5

More than 900 great restaurants, including four Gordon Ramsay favourites from £5

Israel prepares to send tanks and troops into Gaza

Michael Sheen: Frost, Nixon and me

Working mums: So that's how she does it

Detox in style: The best spas on the planet

Salman Rushdie: I won't marry again

Giant killing? Gable to the FA Cup third round

Chancellor on brink of second bailout for banks

Billions may be needed as lending squeeze tightens

99p

图5-1

很多人第一时间想到了初始化配置。没错，在 Fabric 的创世区块中记录的都是通道的配置信息，不仅包含 Fabric 的网络结构配置，更涉及网络中每个节点的身份、认证、权限、交易的流程与限制，等等。在本章中，我们将从解析 Fabric 创世区块开始，深入理解交易与 Chaincode 的原理。

5.1　Fabric 的创世区块

5.1.1　Fabric 的网络结构定义

在 Fabric 的初始化配置中，具体记录了哪些信息呢？

我们回头看看在第 2 章中讲过的并于 Fabric 网络的内容。在一个 Fabric 网络中通常包含 Orderer、Peer 两种节点，根据不同的业务需求，Fabric 网络还能支持多个通道和多个组织。虽然不同通道的创世区块是不同的，但每个通道的组成结构都是相同的。无论是通道

还是组织，Fabric 在设计时就对消息的可见性、权限及安全问题进行了充分考虑。下面举一个例子来说明 Fabric 在权限及安全性方面的考虑。

通常来说，一个交易业务会由若干个不同的功能与参与者组成，例如在电商网站上购买一个商品时，通常会涉及用户、商户、订单、库存、支付、结算、发票、物流、配送、积分、评价、客服、退换货、优惠扣减和来源活动统计等功能，而每个功能的参与者都是不同的，对每个功能的参与者知晓的信息需要有所隔离，因为并不是每个功能的参与者都需要知晓所有交易信息，例如，负责计算优惠扣减的部门不需要知晓用户的详细地址，负责物流配送的部门不需要知晓商品的具体价格，用户也不需要知晓进货成本。如果让信息错误地流通到其他单位，则会造成隐私泄露或者安全问题。

为了从架构上解决这类问题，Fabric 设计了通道和组织，用于划分不同的功能与参与者。我们可以将通道理解为业务中的某项具体功能，组织则是每个功能对应的参与者。在不同的通道之间信息不共享，在不同的组织之间数据与权限也是隔离的。还是用上面的例子来做假设，如果需要使用 Fabric 构建一个电商系统，那么用户、商户、订单、库存、支付等每个功能都需要有一个单独的通道来管理与流通数据，使参与每个通道的组织都只能专注于当前环节的业务，例如，在商户通道中不能读取关于库存的数据，在库存通道中不能读取关于用户的数据。在独立的通道内部，不同组织的权限也是不一样的，例如，用户在下单前可以通过库存通道了解存货情况，但是并没有权限修改库存记录，库存的写入权限只能在商家那里；而对于物流来说，商户和客户只有读取权限，物流公司才有写入权限。

总的来说，通常一个大型的 Fabric 应用网络是由若干个通道组成的，通道是由若干个组织组成的，每个通道都可能代表一种功能组成，每个功能又都有若干组织参与。

那么 Fabric 是如何定义通道和组织的呢？在 Fabric 的配置文件中，用于定义整个网络的基本结构的是 configtx.yaml 文件，我们可以在源码中的 sampleconfig 目录下查看该文件。configtx.yaml 文件被分为 Profiles、Organizations、Orderer、Application、Capabilities 五个段，如下所述。

（1）Profiles 段主要用于定义基础的网络结构，每个 Profile 都是一个通道类型的定义，注意，这里的定义并不包含具体的通道名称，即 Profile 在相似类型的通道定义中是可以重用的，在创建通道时为不同的通道指定不同的名称即可。我们可将通道定义分为两种，一种是系统通道定义，另一种是业务通道定义。首先，我们来看看系统通道定义。在创世区块中定义了 Orderer、Consortiums 与 Application，常见的定义如下：

```
    SampleDevModeSolo:
```

```
    Orderer:
        <<: *OrdererDefaults
        Organizations:
            - <<: *SampleOrg
              AdminPrincipal: Role.MEMBER
    Application:
        <<: *ApplicationDefaults
        Organizations:
            - <<: *SampleOrg
              AdminPrincipal: Role.MEMBER
    Consortiums:
        SampleConsortium:
            Organizations:
                - <<: *SampleOrg
                  AdminPrincipal: Role.MEMBER
```

Orderer 主要用于定义排序节点组织，在 Orderer 段下主要定义 Orderer 的参数及组织，每个组织都会定义自己的 MSP，用于区分不同的组织。

Consortiums 和 Application 段的定义看起来非常相似，都包含了对节点组织的定义，那么，为什么没有将二者放到一起定义呢？Fabric 可能出于对以下几点的考虑。

- 在整个 Fabric 网络启动的时候，Orderer 都会创建一个系统通道，这个系统通道中的所有成员都可以接收到通道创建的事件及通道的定义；如果这个 Fabric 网络中的所有组织都可以接收到这个事件，则会打破 Fabric 对通道信息有所隔离的限制，所以需要限制同步的对象。
- 业务通道中的组织也不能凭空出现，这些组织都需要被事先注册在系统通道对应的联盟中。
- 系统通道需要包含允许创建通道的组织的 MSP 定义，以确定允许创建通道的组织。

因此，Fabric 将执行具体业务的节点组织与执行通道管理的节点组织区分开来。

Consortiums 用于定义所有组织所属的联盟、组织与联盟的隶属关系，以及组织对应创建的权限；Application 用于定义执行业务的节点组织，Application 中的组织隶属于对应的 Consortium。

我们在测试代码中经常看到被写在一起的 Consortiums 与 Application，在实际的应用场景中，二者应该是严格区分的，在系统通道中需要避免出现 Application 的业务组织，在一般的业务场景中也不能定义 Consortiums 的通道管理组织。

一个常见的系统通道定义如下：

```
    SampleSingleMSPSolo:
        Orderer:
            <<: *OrdererDefaults
            Organizations:
                - *SampleOrg
        Consortiums:
            SampleConsortium:
                Organizations:
- *SampleOrg
```

一个常见的业务通道定义如下：

```
    SampleSingleMSPChannel:
        Consortium: SampleConsortium
        Application:
            <<: *ApplicationDefaults
            Organizations:
                - *SampleOrg
```

在这里需要强调的是，与系统通道定义的 Consortiums 不同，业务通道定义的不是 Consortiums，而是 Consortium（少了一个 s），系统会从已经注册的 Consortiums 中匹配同名的联盟，如果没有同名的 Consortiums，则会报错，Application 所定义的组织也必须隶属于声明的 Consortium。这样就完成了 Fabric 网络结构的初步架构设计。

（2）Organizations 段主要用于定义组织，例如：

```
Organizations:
    - &SampleOrg
        Name: SampleOrg
        ID: DEFAULT
        MSPDir: msp
        AdminPrincipal: Role.ADMIN
        AnchorPeers:
            - Host: 127.0.0.1
              Port: 7051
```

在这里定义了一个名为 SampleOrg 的组织，至于在这个组织中有哪些节点，似乎并没有声明。在通常情况下，网络的定义都会包含节点的地址、主机名或者标识符，那么 Fabric 网络是怎样定义其中的节点并识别其身份的呢？答案就是通过在 Organizations 中定义的 MSP。

每个不同的组织都可以有自己的 MSP，MSP 是一套基于 CA 证书的签名体系，其核心是一套 CA 证书，这套 CA 证书可以签发一系列证书，反过来，这一系列证书也可以用该组织的 CA 证书来验证。也就是说，Fabric 网络中的所有数据包都会附带签名，接收方在接收到数据包后会提取签名，然后根据数据包声明的 MSP 提取与 MSP 对应的 CA 证书，即通过 CA 公钥对签名进行解密，如果能够成功解密，则说明这个节点确实归属于它声明的组织，否则说明这个节点并不属于它声明的组织。这样，我们既可以通过数字签名验证数据包是否被篡改或者伪造，也可以通过 CA 签发证书的方式很方便地扩展同一组的其他节点。

ID 字段主要用于定义 MSP 的名称；MSPDir 用于定义 MSP 证书存放的目录；AdminPrincipal 则用于定义该组织管理员策略的主题类型，也就是定义组织配置需要由 Admin 对象来管理还是由 Member 对象来管理；AnchorPeer 则涉及接下来会讲到的 Gossip 网络。

这里对 Gossip 网络做个简单介绍。想象一下，Fabric 网络是一所学校，每个组织都是一个班级，每个班级都有若干学生（节点），学生之间通过悄悄话（Gossip）沟通，而班级与班级之间如果完全通过悄悄话来沟通，则很容易导致消息的不一致。为了让班级内的消息一致，老师指定了一个班长，统一接收外来的消息并告诉班里的同学，这个班长就是 AnchorPeer。

（3）Orderer、Application 与 Capabilities 段分别用于定义排序节点的属性、应用节点的属性及网络的兼容性，配置内容相对明确，这里不再赘述。

现在，我们已经知道 configtx.yaml 配置文件通过定义 Fabric 网络中的通道、组织和成员来确定整个网络的组织结构，下面对创世区块的结构进行解析，来看看这些信息是如何被打包、传递及使用的。

5.1.2　创世区块的结构

Fabric 的创世区块是何时产生的呢？在第 3 章中提到了，Orderer 节点会在启动的过程中创建默认通道（testchainid）时生成创世区块，并在创建通道时根据通道的配置生成不同的创世区块。我们可以通过以下命令获取 Orderer 节点创建的创世区块：

```
[root@localhost fabric]# build/bin/peer channel fetch oldest genesis.block -c
testchainid -o 127.0.0.1:7050
```

实际上，通过 Configtxgen 工具也能生成指定通道的创世区块：

```
[root@localhost fabric]# build/bin/configtxgen -profile SampleDevModeSolo
-outputBlock genesis.block
```

在获取创世区块后，我们可以通过 Configtxgen 工具解析区块的内容：

```
[root@localhost fabric]# build/bin/configtxgen -inspectBlock genesis.block
```

或者直接在 Orderer 节点启动时添加环境变量或者修改设置，直接让区块以 JSON 格式写入文件中：

```
[root@localhost fabric]# ORDERER_GENERAL_LEDGERTYPE=json
ORDERER_GENERAL_GENESISPROFILE=SampleDevModeSolo build/bin/Orderer
```

限于篇幅，这里不再详细列举区块的实际内容，按这些步骤提取出来的内容都非常相近。

1. 区块结构定义

我们首先从某个 JSON 对象的结构开始，梳理创世区块的结构，然后根据代码进行分析。这个 JSON 对象被分为三部分，如图 5-2 所示。

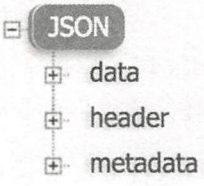


图5-2

这是一个标准的区块（Block）结构，定义区块的代码在 protos/common/ common.pb.go 中：

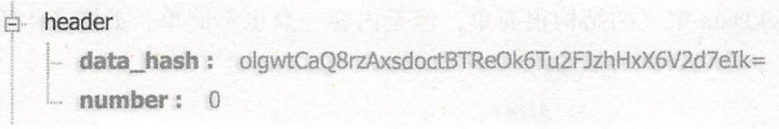
```
type Block struct {
 Header   *BlockHeader
 Data     *BlockData
 Metadata *BlockMetadata
}
```

Golang 对自家的序列化协议 ProtoBuf 有很好的支持，所以 Golang 的数据结构能被轻松地序列化或者转换成可读性很高的 JSON 格式。在这个 Block 中又包含了 BlockHeader、

BlockData 与 BlockMetadata 三个结构，我们先来看看 BlockHeader 的定义：

```
type BlockHeader struct {
 Number       uint64
 PreviousHash []byte
 DataHash     []byte
}
```

在 BlockHeader 中包含三个对象：一个无符号的 64 位整数 Number，以及两个字节数组 PreviousHash 与 DataHash。Number 是区块的序号，PreviousHash 是前一个区块的哈希值，DataHash 是当前区块 Data 部分的哈希值。我们将这个 JSON 对象的 header 对象展开后可以看到区块序号（number）为 0，data 部分的哈希值（data_hash）是一串被 Base64 压缩过的字串，但并没有看到 PreviousHash，这是因为作为创世区块，前一个区块值默认为空，而 Configtxgen 工具在导出时会根据字段属性“omitempty”忽略空字段。



BlockMetadata 主要记录不需要固化在 BlockData 中的数据：

```
type BlockMetadata struct {
    Metadata          [][]byte
}
```

在定义中，BlockMetadata 主要记录一个二维的字节数组，而且这部分信息并不作为 BlockData 的一部分参与哈希运算，那么哪些数据可以作为 Metadata 被记录呢？通过查看源码，我们可以找到以下定义：

```
const (
    BlockMetadataIndex_SIGNATURES          BlockMetadataIndex = 0
    BlockMetadataIndex_LAST_CONFIG         BlockMetadataIndex = 1
    BlockMetadataIndex_TRANSACTIONS_FILTER BlockMetadataIndex = 2
    BlockMetadataIndex_ORDERER              BlockMetadataIndex = 3
)
```

BlockMetadata 主要包含四个索引，分别是 Signatures、Last Config、Transactions Filter 和 Orderer，这意味着如果需要让 BlockMetaData 支持更多类型的 MetaData，则需要对相关类型进行扩张。Signatures 顾名思义是区块签名；Last Config 则指生成这个区块时的最新配置区块序列号；Transactions Filter 是区块中的有效交易过滤器；Orderer 记录排序时的

参数，暂时只记录 Kafka 打包时的偏移量。如图 5-3 所示，这些内容在创世区块中都为空，因此对这部分内容暂时一笔带过。

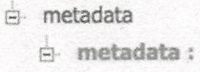


图5-3

2. BlockData 的结构定义

接下来就是上面所述内容的重点了。BlockData 记录了所有交易及配置信息：

```
type BlockData struct {
    Data      [][]byte
}
```

虽然 BlockData 定义的结构很简单，但是内容一点也不简单，如图 5-4 所示。

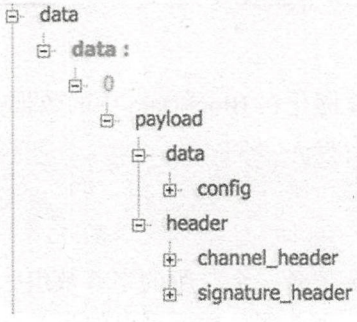


图5-4

在 Data 的二维数组中存储的是序列化后的 payload 对象数组，payload 通常被称为有效载荷或者负载，在每个 payload 中都会保存 Header 与 Data：

```
type Payload struct {
    Header *Header
    Data []byte
}
```

Data 主要用于存放交易数据，Header 主要用于存放交易标识信息，以及为了防止重放攻击而记录的额外数据。重放攻击（Replay Attack）指的是一种网络攻击，通过恶意、欺诈性地重复或拖延正常的数据传输来实施，因工作原理如同重放歌曲一样而得名。这也

就意味着，攻击方即使并不知道数据包的具体消息结构或者如何解密，也可能通过简单地再次发送数据包导致系统数据混乱，因此，需要为每笔交易数据添加唯一标识来防止这种情况的发生。那么，具体在 Header 中添加了哪些信息用于标识数据的唯一性呢？后面会进行详细讲解。

3. Header 的定义与防重放攻击

Header 的定义如下：

```
type Header struct {
    ChannelHeader      []byte
    SignatureHeader    []byte
}
```

我们可以看到，在 Header 中主要包含序列化后的 ChannelHeader 与 SignatureHeader。SignatureHeader 的定义比较简单：

```
type SignatureHeader struct {
    Creator []byte
    Nonce []byte
}
```

Creator 是序列化后的消息创建者的身份认证，Nonce 是唯一的随机数。了解比特币的人可能会很熟悉这个随机数，在比特币区块中需要通过猜测 Nonce 值来生成符合规则的 SHA256 值，在这里并没有如此耗费算力的计算，主要通过 Golang 提供的 crypto/rand 接口来生成（参见 common/crypto/random.go）。ChannelHeader 的定义稍微复杂一点：

```
type ChannelHeader struct {
    Type int32
    Version int32
    Timestamp *google_protobuf.Timestamp
    ChannelId string
    TxId string
    Epoch uint64
    Extension []byte
}
```

ChannelHeader 中的 Type 是当前 Payload 的类型值，定义如下：

```
const (
    HeaderType_MESSAGE                  HeaderType = 0
```

```
    HeaderType_CONFIG                    HeaderType = 1
    HeaderType_CONFIG_UPDATE             HeaderType = 2
    HeaderType_ENDORSER_TRANSACTION      HeaderType = 3
    HeaderType_ORDERER_TRANSACTION       HeaderType = 4
    HeaderType_DELIVER_SEEK_INFO         HeaderType = 5
    HeaderType_CHAINCODE_PACKAGE         HeaderType = 6
)
```

MESSAGE 用于标识已签名但不可见的交易，在当前版本中没有用到；CONFIG 用于标识通过 Endorser 提交的或者 Orderer 节点自己生成的通道配置交易；CONFIG_UPDATE 用于标识通过广播提交的通道配置交易；ENDORSER_TRANSACTION 用于标识由背书节点提交的交易；ORDERER_TRANSACTION 用于标识 Orderer 节点在接收到广播之后内部处理的通道配置交易；DELIVER_SEEK_INFO 用于标识 Deliver API 查找消息的类型；CHAINCODE_PACKAGE 用于标识 Chaincode 的安装包。

ChannelHeader 中的 Version 是当前 Payload 的消息版本；Timestamp 是当前 Payload 的时间戳，时间戳由客户端或者 SDK 在提交时生成；Channle_id 是提交的通道 ID；Extension 用于保存扩展信息，这些都是常用的配置信息。若要实现 Header 的基本动机“防止重放攻击”，则需要依赖以下两个参数。

- Epoch：代表当前交易在生成时的区块高度，如果在同一 Epoch 下相同的消息出现了两次或者以上，则交易会被拒收；在设计时考虑到用于判断交易提案的 Epoch 值与当前区块高度的差异，将来也可以实现拒收交易。
- TxId：代表交易的唯一 ID，在交易生成时生成，Endorser 与 Committer 都会对这个值的唯一性和有效性进行判定。

Header 部分的内容如图 5-5 所示。

总之，Header 为了防止重放攻击，通过 TxId、Epoch 与签名部分的 Nonce 来标识单个交易的唯一性，避免相同的交易被提交两次。需要注意的是，重放攻击与双花攻击不同，重放攻击通过提交完全相同的包来达到攻击的目的，双花攻击则利用逻辑上的漏洞将同一笔钱支付 N 次。事实上，Fabric 对这两种情况都做了很好的处理。双花攻击的预防机制在之后的章节中会提到。

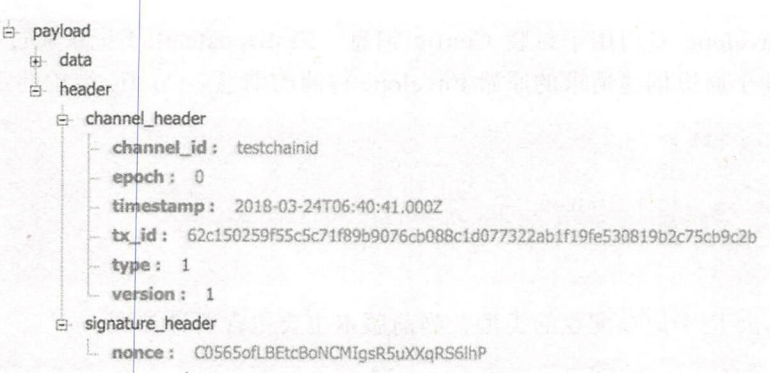


图5-5

5.1.3　创世区块的通道定义

接下来我们看一下真正的内容，即 Payload 中 Data 的内容，如图 5-6 所示。

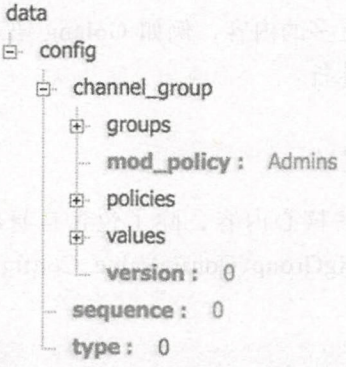


图5-6

1. ConfigEnvelope 的结构定义

Data 主要保存的是一个序列化后的字节数组，在这里保存的是一个序列化后的 ConfigEnvelope 结构：

```
type ConfigEnvelope struct {
    Config    *Config
    LastUpdate *Envelope
}
```

ConfigEnvelope 专门用于封装 Config 消息，LastUpdate 用于记录通过 ConfigUpdate 消息发送的并于通道创建请求的原始 Envelope 封装的消息。Config 结构的定义如下：

```
type Config struct {
 Sequence       uint64
 ChannelGroup   *ConfigGroup
 Type           int32
}
```

ConfigType 用于记录配置的类型，当前版本主要包含两种类型：

```
const (
    ConfigType_CHANNEL     ConfigType = 0
    ConfigType_RESOURCE    ConfigType = 1
)
```

ConfigType_CHANNEL 代表包内容为通道配置参数，在以上创世区块示例中，ConfigType 为 0，代表在 ConfigGroup 中存储的是通道配置；ConfigType_RESOURCE 代表包内容为通道资源，这是 Fabric 1.1 版本新增的定义。Fabric 计划在接下来的版本里增加动态配置选项，可以承载更多的内容，例如 Golang 语言的 API、Chaincode 参数等，为之后更复杂的版本配置做准备。

2. ChannelGroup 结构定义

ChannelGroup 作为配置的核心内容，除了包含自身的版本（Version）与修改策略（ModPolicy），还主要包含 ConfigGroup、ConfigValue、ConfigPolicy 的 Map 对象，即 Groups、Values 和 Policies：

```
type ConfigGroup struct {
    Version        uint64
    Groups         map[string]*ConfigGroup
    Values         map[string]*ConfigValue
    Policies       map[string]*ConfigPolicy
    ModPolicy      string
}
```

ConfigGroup 作为配置的基础定义结构，在这里是以递归形式组织的，也就是说，ConfigGroup 可以组织一个树状的配置结构，每一层都可以有自己的 Groups、Values 与 Policies，如图 5-7 所示。

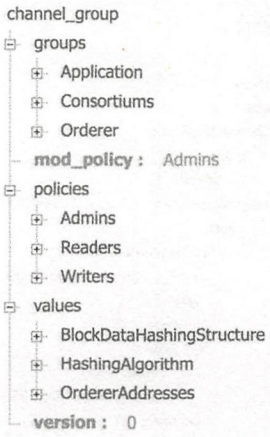


图5-7

作为根部的 ConfigGroup 名为 channel_group，定义了整个通道的配置，我们通过观察它的内容可以发现，之前提到的在 configtx.yaml 中定义的网络结构就被存储在这里，包括 Orderer、Consortiums、Application 及它们对应的配置。因为 Orderer、Consortiums 与 Application 都可能是多级结构，每一级都可能有自己的配置项、管理策略及子结构，所以采用了 ConfigGroup 的递归方式来定义网络架构。Groups 部分定义了当前的网络结构，我们可以看到它包含了 Orderer、Consortiums 与 Application 三部分。

Values 部分主要定义了 BlockDataHashingStructure、HashingAlgorithm 与 OrdererAddresses。HashingAlgorithm 是配置指定的哈希算法；OrdererAddresses 是 Orderer 节点的地址与端口；BlockDataHashingStructure 用于指定在计算 BlockData 的哈希值时使用的 Merkle Tree 的宽度。到 Fabric 1.1 版本为止，暂时没有使用 Merkle Tree BlockData 的哈希值进行计算，而是在简单地将数组拼接后进行哈希运算。

Policies 部分用于定义当前 Group 的管理、写入及读取策略。Group 与 Policies 主要涉及 MSP 管理与策略管理，基于 MSP 的成员管理与签名校验的策略管理部分也是 Fabric 的特色。

3. Consortium 策略定义

我们在配置中经常可以看到 version 与 mod_policy，在 channel_group 下的配置项目中都包含这两个元素，这两个元素主要用于管理通道配置的更新与升级。mod_policy 决定了需要由谁修改，在大多数情况下是 Admins 也就是本组织的管理员可以修改，但 Consortiums

部分的情况比较特殊，如图 5-8 所示。

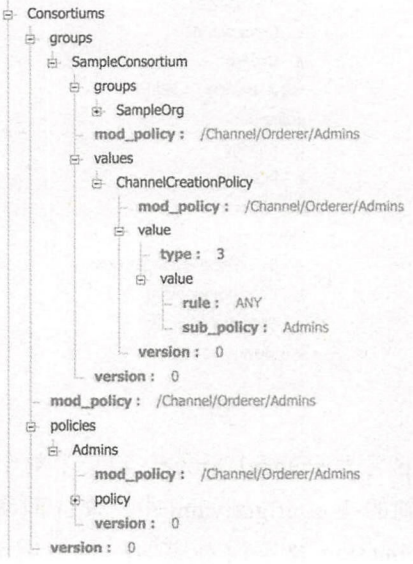


图5-8

可以看到，Consortiums、Consortiums/policy、SampleConsortium 与 ChannelCreationPolicy 的 mod_policy 都是/Channel/Orderer/Admins，这也就意味着在当前通道的 Orderer 集群中拥有 Admin 身份的节点签名的配置才能修改这些属性。而每个配置进行的每一次更新都会让 Version 版本号加 1，这也是从安全角度进行考虑的。设想一下，如果两个拥有相同权限的节点对同一个配置进行修改，那么以谁的为准？是以先提交的为准还是以后覆盖的为准？在这种情况下不可避免地会产生一些争议，Fabric 的解决方案是在进行每一次提交时，都要把已有的需要修改的配置项，与修改后的配置项一起打包提交，在已有的需要修改的配置项与当前存储的配置项版本相同之后，才能引入和更新配置。已有的配置项在 Fabric 中被记为 ReadSet，修改后的配置项被记为 WriteSet。是不是觉得 ReadSet 与 WriteSet 很眼熟？没错，在 3.1.3 节中，我们通过 Configtxgen 工具生成了一个通道配置交易，其中就包含 ReadSet 与 WriteSet 的示例，若有兴趣，则可以仔细研究一下。

5.1.4 创世区块的生成代码解析

现在，我们已经了解了创世区块的内容与组成，接下来了解系统通道与业务通道的创世区块的生成方法与处理流程。

创世区块的生成方法有两种，一种是由 Orderer 节点生成，代码在 orderer/common/server/main.go 中；一种是由 Configtxgen 工具生成，代码在 common/tools/configtxgen/main.go 中。虽然有两个入口，但实际上调用的是同一种方法，也就是 common/tools/configtxgen/encoder/encorder.go 中的 New()方法。不管是由 Orderer 节点生成还是由 Configtxgen 工具生成，都需要在加载 configtx.yaml 中对应的 profile 后，作为参数传递给 encoder.New()：

```
func New(config *genesisconfig.Profile) *Bootstrapper {
    channelGroup, err := NewChannelGroup(config)
    …
    return &Bootstrapper{
        channelGroup: channelGroup,
    }
}
```

1. NewChannelGroup

在 encorder.New 函数中只通过调用 NewChannelGroup 生成了一个对象，在解析创世区块时，在 Payload 中记载的对象就是一个 channel_group。我们再进入 NewChannelGroup 中进行查看：

```
func NewChannelGroup(conf *genesisconfig.Profile) (*cb.ConfigGroup, error) {
  // 必须初始化配置中的Orderer定义
    if conf.Orderer == nil {
        return nil, errors.New("missing orderer config section")
    }

channelGroup := cb.NewConfigGroup()
// 添加隐式元策略，因为channel_group自身并没有实体对象，所以它的修改策略依赖于自身包含的组织的
// 策略评估结果，隐式元策略分为ANY、MAJORITY、ALL三种，这意味着需要内部组织任意一个、两个或
// 三种评估才能生效
addImplicitMetaPolicyDefaults(channelGroup)
// 将默认的哈希算法bccsp.SHA256打包成value后添加到channelGroup中
addValue(channelGroup, channelconfig.HashingAlgorithmValue(),
channelconfig.AdminsPolicyKey)
// 将Mekle tree的宽度MaxUint32打包成value后添加到channelGroup中
addValue(channelGroup, channelconfig.BlockDataHashingStructureValue(),
channelconfig.AdminsPolicyKey)
// 将在配置中写入的Orderer地址打包成value后添加到channelGroup中
addValue(channelGroup,
```

```
channelconfig.OrdererAddressesValue(conf.Orderer.Addresses),
ordererAdminsPolicyName)
    // 如果当前所在的联盟不为空，则将当前联盟打包后添加到channelGroup中
     if conf.Consortium != "" {
            addValue(channelGroup, channelconfig.ConsortiumValue(conf.Consortium),
channelconfig.AdminsPolicyKey)
    }
    // 将conf.Capabilities功能声明打包成value后添加到channelGroup 中
    if len(conf.Capabilities) > 0 {
            addValue(channelGroup, channelconfig.CapabilitiesValue(conf.Capabilities),
channelconfig.AdminsPolicyKey)
    }
    …
    // 对于创世区块来说，OrdererGroup是必需的
    channelGroup.Groups[channelconfig.OrdererGroupKey], err =
NewOrdererGroup(conf.Orderer)
    if err != nil {
            return nil, errors.Wrap(err, "could not create orderer group")
    }
    // 注意此处，在创世区块中Application与Consortiums都不是必需的
    if conf.Application != nil {
            channelGroup.Groups[channelconfig.ApplicationGroupKey], err =
NewApplicationGroup(conf.Application)
    …
    if conf.Consortiums != nil {
            channelGroup.Groups[channelconfig.ConsortiumsGroupKey], err =
NewConsortiumsGroup(conf.Consortiums)
    …
    }
    // 定义channel_group的修改策略为Admins，也就是只有Admin证书的成员可以修改内部的值
    // 注意，这个修改者的证书应该能够通过隐式元策略的评估
    channelGroup.ModPolicy = channelconfig.AdminsPolicyKey
    return channelGroup, nil
    }
```

2. NewOrdererGroup

我们注意到，在 NewChannelGroup 中还调用了 NewOrdererGroup、NewApplicationGroup 与 NewConsortiumsGroup。我们先从 NewOrdererGroup 开始梳理这部分逻辑：

```
    func NewOrdererGroup(conf *genesisconfig.Orderer) (*cb.ConfigGroup, error) {
    ordererGroup := cb.NewConfigGroup()
    // 添加隐式元策略
    addImplicitMetaPolicyDefaults(ordererGroup)
    // BlockValidation区块校验策略作为Orderer节点的通用策略，在这里也是作为隐式元策略定义的，
    // 因为ordererGroup自身并没有实体
    ordererGroup.Policies[BlockValidationPolicyKey] = &cb.ConfigPolicy{
            Policy:
policies.ImplicitMetaAnyPolicy(channelconfig.WritersPolicyKey).Value(),
            ModPolicy: channelconfig.AdminsPolicyKey,
    }
    // 将共识类型打包成value后添加到ordererGroup中
    addValue(ordererGroup, channelconfig.ConsensusTypeValue(conf.OrdererType),
channelconfig.AdminsPolicyKey)
    // 将区块批处理的相关参数打包成value后添加到ordererGroup中
    addValue(ordererGroup, channelconfig.BatchSizeValue(
            conf.BatchSize.MaxMessageCount,
            conf.BatchSize.AbsoluteMaxBytes,
            conf.BatchSize.PreferredMaxBytes,
    ), channelconfig.AdminsPolicyKey)
    // 将区块批处理时的长参数打包成value后添加到ordererGroup中
    addValue(ordererGroup,
channelconfig.BatchTimeoutValue(conf.BatchTimeout.String()),
channelconfig.AdminsPolicyKey)
    // 将通道数量限制打包成value后添加到ordererGroup中
    addValue(ordererGroup, channelconfig.ChannelRestrictionsValue(conf.MaxChannels),
channelconfig.AdminsPolicyKey)
    // 将conf.Capabilities功能声明打包成value后添加到ordererGroup中
    if len(conf.Capabilities) > 0 {
            addValue(ordererGroup, channelconfig.CapabilitiesValue(conf.Capabilities),
channelconfig.AdminsPolicyKey)
    }
    // 如果共识算法是Kafka，则将Kafka集群的断路器配置打包成value后添加到ordererGroup中
    switch conf.OrdererType {
        case ConsensusTypeSolo:
        case ConsensusTypeKafka:
            addValue(ordererGroup,
channelconfig.KafkaBrokersValue(conf.Kafka.Brokers), channelconfig.AdminsPolicyKey)
    …
    }
    // 将在Orderer下定义的组织添加到ordererGroup中
```

```
for _, org := range conf.Organizations {
        var err error
        ordererGroup.Groups[org.Name], err = NewOrdererOrgGroup(org)
        …
}
…
```

NewOrdererGroup 最后生成的 Orderer 配置结构如图 5-9 所示。

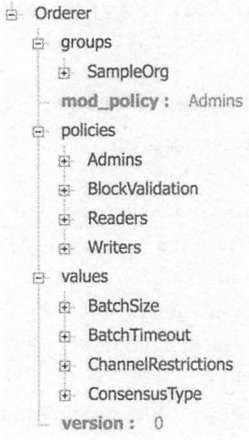


图5-9

在 NewOrdererGroup 中主要定义了所有通道通用的 Orderer 属性，以及 BlockValidation 区块验证策略。需要注意的是，BlockValidation 作为 Orderer 组织的策略，最好与 Application 组织区分开来，以便 Peer 节点在收到区块时确定这是由 Orderer 节点签名的。

如果 Orderer 组织与 Application 组织相同，则我们可以试想一种情况，如果一个 Application 节点伪造了一个区块，使用 Orderer 与 Application 共用的组织进行签名，然后通过 Gossip 协议将区块传递到网络中，其他节点就无法区分这个区块是否真正由 Orderer 节点生成，整个区块链会失去一致性和同步性。因此在生产环境中，需要严格区分 Orderer 组织与 Application 组织。同理，在整个 Fabric 的架构设计环节中，对安全的考虑是第一位的，我们在修改代码或者设计区块链应用时，也需要着重考虑这一点。BlockValidation 作为隐式元策略之一，默认设置的规则为 Any，也就是说在进行区块有效性策略验证时，只要 Orderer 包含的组织的签名之一生效，即可通过。

5.1.5　组织与策略的定义

接下来，我们再以 ordererGroup.Groups 对象中的 SampleOrg 为例，解读组织的定义方式。在 SampleOrg 中，除了配置节点必需的 mod_policy 与 version，主要包含 policies 与 values 定义，如图 5-10 所示。

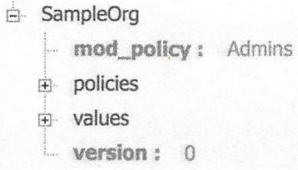


图5-10

在代码中，我们可以看到：

```
    func NewOrdererOrgGroup(conf *genesisconfig.Organization) (*cb.ConfigGroup, error)
{
    mspConfig, err := msp.GetVerifyingMspConfig(conf.MSPDir, conf.ID, conf.MSPType)
        …
    ordererOrgGroup := cb.NewConfigGroup()
    // 添加签名策略
    addSignaturePolicyDefaults(ordererOrgGroup, conf.ID, conf.AdminPrincipal !=
genesisconfig.AdminRoleAdminPrincipal)
    // 将MSP配置打包成value后放入ordererOrgGroup中
    addValue(ordererOrgGroup, channelconfig.MSPValue(mspConfig),
channelconfig.AdminsPolicyKey)
    …
    }
```

其中，addSignaturePolicyDefaults 用于添加一个默认的签名策略，addValue 用于添加 MSP 配置，MSP 机制和签名策略共同构建了 Fabric 中复杂而又精巧的安全认证体系。对于 MSP 机制，会在第 6 章中详细讲解。签名策略相当于用户在提交某个操作时对具体的操作类型进行分类，即读、写、修改配置；不同的操作要求不同的签名策略，读对应 Reader 策略，写对应 Writer 策略，通道配置修改对应 Admin 策略，每个策略对签名的要求都不一样，但基本上都是通过 msp_identifier 与 role 来决定的，msp_identifier 指使用哪个 MSPID，role 指签名证书代表的身份类型。在 MSP 下有 Admincerts 与 CAcerts 两种证书，由 Admincerts 签发的证书代表 Admin 身份，而由 CAcerts 签发的证书代表成员身份，这也就意味着我们可以使用 Admin 身份做任何事情，而成员身份在大多数情况下只能进行读写操

作。同时，在默认的签名策略中，要求签名数量符合 NOutOf(1)的规则，也就是前面所说的 ANY 规则，即只需有一个签名是生效的。

上层的隐式元策略在评估时会逐层递归地进行策略评估，最终执行的是每个子对象的签名策略，签名策略的评估结果在上层汇聚后再根据策略判断生成统一的评估结果。签名策略自身也可以进行多种方式的组合，例如：

```
SignaturePolicyEnvelope{
   version: 0,
   policy: SignaturePolicy{
      n_out_of: NOutOf{
         N: 2,
         policies: [
            SignaturePolicy{ signed_by: 0 },
            SignaturePolicy{
               n_out_of: NOutOf{
                  N: 1,
                  policies: [
                     SignaturePolicy{ signed_by: 1 },
                     SignaturePolicy{ signed_by: 2 },
                  ],
               },
            },
         ],
      },
   },
   identities: [mspP1, mspP2, mspP3],
}
```

在进行策略评估时，有 MSPID 为 mspP1、mspP2、mspP3 的三个组织，若通过 signed_by 指定其中的某个组织，那么这个复杂的写法要求的评估条件是 mspP2、mspP3 两个的签名之一与 mspP1 的签名，只有满足这个条件才能通过策略评估。当然，这些复杂的策略需要我们根据需求，自己调用 Fabric 提供的方法实现。Fabric 提供的默认策略如下。

- /Channel/Readers：隐式元策略，要求提案签名能通过 Channel 下至少一个组织的 Readers 策略评估。
- /Channel/Writers：隐式元策略，要求提案签名能通过 Channel 下至少一个组织的 Writers 策略评估。
- /Channel/Admins：隐式元策略，要求提案签名能通过 Channel 下大多数组织的 Admins

策略评估。

- /Channel/Application/Readers：隐式元策略，要求提案签名能通过 Channel/ Application 下至少一个组织的 Readers 策略评估。
- /Channel/Application/Writers：隐式元策略，要求提案签名能通过 Channel/ Application 下至少一个组织的 Writers 策略评估。
- /Channel/Application/Admins：隐式元策略，要求提案签名能通过 Channel/ Application 下大多数组织的 Admins 策略评估。
- /Channel/Orderer/Readers：隐式元策略，要求提案签名能通过 Channel/Orderer 下至少一个组织的 Readers 策略评估。
- /Channel/Orderer/Writers：隐式元策略，要求提案签名能通过 Channel/Orderer 下至少一个组织的 Writers 策略评估。
- /Channel/Orderer/Admins：隐式元策略，要求提案签名能通过 Channel/Orderer 下大多数组织的 Admins 策略评估。
- /Channel/*/Org/Readers：签名策略，定义每个组织的 Readers 签名策略与 MSP 之间的关联。
- /Channel/*/Org/Writers：签名策略，定义每个组织的 Writers 签名策略与 MSP 之间的关联。
- /Channel/*/Org/Admins：签名策略，定义每个组织的 Admins 签名策略与 MSP 之间的关联。

至此，我们对创世区块的内容有了较为深入的了解。创世区块作为区块的一种类型，具有大多数区块的属性，我们可以看到区块标志性的 DataHash、PreviousHash、Epoch 等属性，也可以看到 Fabric 用于防重放攻击的 Nonce 等设置；作为 Fabric 网络配置的第 1 个区块，创世区块承载了网络初始化时的所有设置，包括组织结构、MSP、隐式元策略、签名策略等内容，以及这些设置的更新配置方法。接下来，我们看一下 Fabric 最重要的环节——交易背后隐藏的逻辑与技术。

5.2　Peer 客户端发起交易

在 Fabric_Demo 的实例中，我们通过 peer chaincode invoke 命令发起了交易，然后通过 peer chaincode query 命令查询交易结果。对于交易在命令背后是如何产生、传输、认证、存储的，以及有效的交易是如何在众多节点之间传递的，本节将从代码层面深入讲解。

Fabric 是一个规模相当大的程序，截至 1.1 版本，Fabric 已经有将近 38 万行 Golang 代码、2000 多个文件，所以要梳理清楚其中的逻辑，并不是一件简单的事情。为了方便大家阅读与理解，我们将代码的关键方法和调用层级整理成了图表，表中的每个函数/方法的表示如图 5-11 所示。

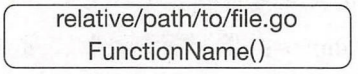


图5-11

在如图 5-11 所示的方块内，第 1 行记录了当前函数所在的文件及相对路径；第 2 行记录了函数或者方法的名称。为了简化录入，大部分方法都没有记录方法接收者，只在有可能混淆的地方注明了接收者的类型，同时没有记录参数或者返回值。为了展示函数之间的调用关系，我们使用了简化的调用栈，函数按照从上往下的方式执行，每一级函数调用都会往右深入一级，如图 5-12 所示。

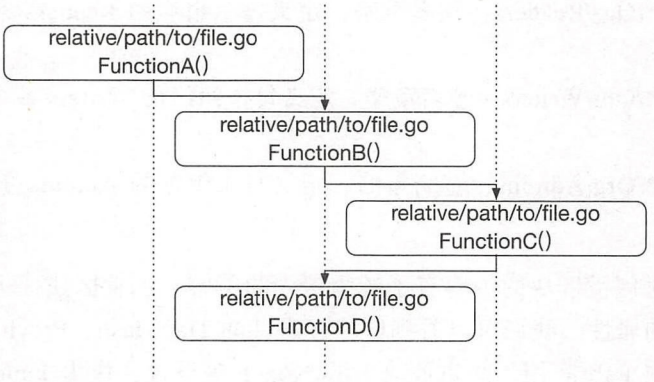


图5-12

可看到，在 FunctionA 内部顺序调用了 FunctionB 与 FunctionD，在 FunctionB 内部调用了 FunctionC。需要注意的是，为了使流程和函数方便查找，流程图忽略了判断条件与接口类，这样在实际配置项有区别时会进入不同的流程，不会完全按照流程图上的步骤来进行。

5.2.1 提案打包

回归正题，Peer 作为 CLI 发起交易时，主要是通过 peer chaincode invoke 指令进行的。

通过代码不难看到，peer chaincode invoke 指令主要通过 peer/chaincode/common.go 文件中的 ChaincodeInvokeOrQuery 函数进行处理，再将 Chaincode 相关的信息（名称、版本及启动参数）打包成 invocation，在根据 signer.Serialize 生成签名 creator 后，将参数传递给 CreateChaincodeProposalWithTransient 打包成提案。注意，TransientMap 仅仅出现在提案环节，在设计上用于存储应用层加密的加密材料，并不会出现在最终的交易和区块中。在 5.1 节讲到，每个交易为了防止重放攻击，需要唯一的 TxID 及 Nonce，而 GetRandomNonce 及 ComputeProposalTxID 用于生成对应的 Nonce 与 TxID，在生成对应的信息之后，通过 CreateChaincodeProposalWithTxIDNonceAndTransient 函数将所有的信息打包成 peer.Proposal。

5.2.2　提案签名

peer.Proposal 就是原始的交易提案，但这个提案还不能马上提交，因为其他 Peer 节点还无法证实该提案真的就是这个节点提交的，很有可能被伪造。接下来是防止伪造的关键环节。在原始提案被提交给 GetSignedProposal 函数后，该函数先通过 GetBytesProposal 将 peer.Proposal 序列化，然后将序列化后的字符串数组提交给 signingidentity.Sign 函数进行签名。与在之前的章节中提到的签名流程一样，在通过 bccsp 的哈希运算提取字符串数组的哈希值之后，Sign 函数使用私钥对哈希值进行签名，将签名信息及序列化的 peer.Proposal 打包成 peer.SignedProposal，在 ProcessProposal 中将提案通过 grpc.invoke 的方式提交给 Endorser，获取背书结果。通过查询 gRPC 的构造过程不难发现，grpc.invoke 使用的 EndorserClient 地址是由 CORE_PEER_ADDRESS 指定的 Peer 地址。

5.2.3　提案背书

如果 Endorser 返回背书结果且没有报错，则 Peer 客户端会在 CreateSignedTx 中将提案重新打包。首先通过 GetBytesProposalPayloadForTx 将 Endorser 返回的 PayloadVisibility 与原有提案的 Payload 打包，也就是说 Endorser 在将来的版本中可以通过 PayloadVisibility 使交易的一部分内容不可见，当前版本只支持全部可见，在通过层层封装打包签名之后生成 common.Envelop 类型，然后使用 BroadcastClient 的 Send 接口将消息发送给 Orderer 节点。

Orderer 节点处理返回的 ProposalResponse，在输出必要的信息后，Peer 客户端的使命就结束了，如图 5-13 所示。

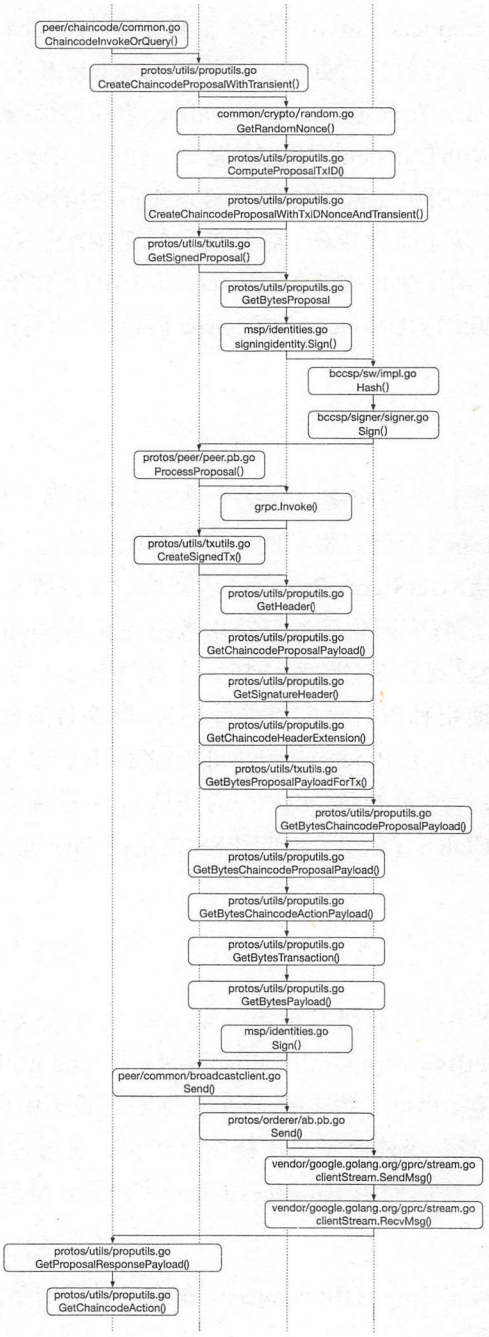


图5-13

5.3　Chaincode 的初始化

在 Fabric 中，Chaincode 可谓灵魂一般的存在：如果没有 Chaincode 的支持，Fabric 就只能是一个速度很慢的分布式账本了，并且 Chaincode 的存在大大扩展了 Fabric 的应用范围。作为区块链 2.0 最重要的特征之一，Chaincode 不仅是一段在 Fabric 之外执行的逻辑，还涉及很多安全方面的考虑：Chaincode 不能直接读写账本，在运行环境下不能被第三方侵入，Peer 节点与 Chaincode 之间的通信不能被第三方窃听或者篡改，等等。基于这些考虑，在 Fabric 中有一套复杂的系统用于确保 Chaincode 的安全。下面通过 Chaincode 的初始化代码来了解与 Chaincode 相关的整套机制。

Peer 节点在启动时会加载系统 Chaincode，如下所述。

- CSCC(Configuration System Chaincode, 配置系统链码), 用于处理通道配置信息。
- LSCC (Lifecycle System Chaincode，生命周期系统链码)，用于处理 Chaincode 生命周期。
- ESCC (Endorsement System Chaincode，背书系统链码)，用于处理交易提案背书签名。
- VSCC (Validation System Chaincode，验证系统链码)，用于处理交易验证，包括检查背书策略和多版本并发控制。
- QSCC (Query System Chaincode，查询系统链码)，用于处理账本查询请求。

这些系统 Chaincode 的初始化在 peer/node/start.go 的 serve 中进行，初始化分为两部分，一部分是自身 ChaincodeServer 的启动，另一部分是 Chaincode 的启动。

5.3.1　ChaincodeServer 的初始化

在初始化 ChaincodeServer 之前，Peer 节点生成了一个自签名证书，该证书主要用于生成 TLS 所需的加密材料，加密 Peer 节点与 Chaincode 之间的通信。在 createChaincodeServer 中通过 NewGRPCServer 建立了与 Chaincode 之间通信的 gRPC 服务；然后在 registerChaincodeSupport 中生成了一个 ChaincodeSupport 对象，将 Chaincode 的运行状态、CA 证书、证书生成器、地址、心跳间隔、启动超时时长等放到 ChaincodeSupport 对象中。在 RegisterSysCCs 中将系统 Chaincode 注册到 inprocContainer 中。这里需要提一下 inprocContainer，在 Fabric 中包含两种容器：一种是大多数人都了解的 Docker 容器；另一种是系统 Chaincode 专用的 inprocContainer。inprocContainer 简单说来是使用一个线程专

门运行 Chaincode，而 Chaincode 的 Docker 容器是通过封装以后的 Docker API 来操作的，虽然两者天差地别，但 Fabric 将两者的接口抽象后通过 VMController 来进行操作，因此在大多数操作流程中都可以认为二者是一致的。考虑到系统 Chaincode 与用户 Chaincode 同样可以被添加、删除和修改，所以 Fabric 在设计上也要求系统 Chaincode 和用户 Chaincode 走同样的初始化与启动流程。在注册完成后将 ChaincodeServer 启动到一个线程中，如图 5-14 所示。

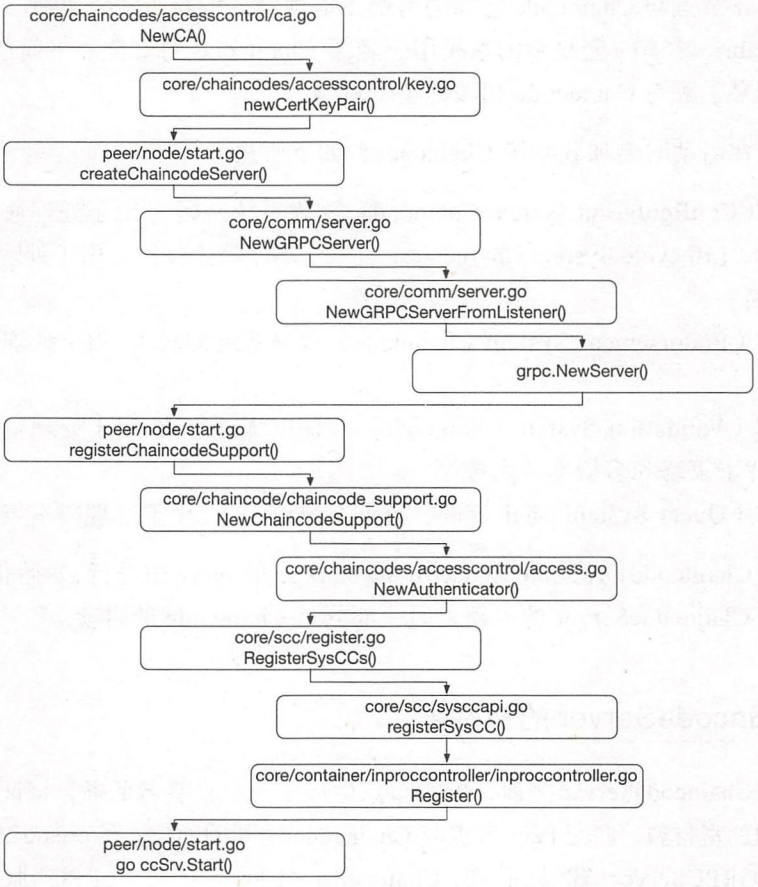


图5-14

5.3.2 通过 initSysCCs 启动容器

在初始化 ChaincodeServer 之后，Peer 节点会在 initSysCCs 函数中初始化系统 Chaincode。

在 DeploySysCCs 的调用中我们可以看到一个空的字符串参数""，这个参数主要用于标识通道 ID，这也就意味着每个通道都可以有自己的系统 Chaincode，但在相同的节点下，每个通道的系统 Chaincode 配置都是相同的，这一点需要注意。在构建 ChaincodeProvider 并填入必要的信息后，通过 ExecuteWithErrorFilter 启动 Chaincode，如图 5-15 所示。

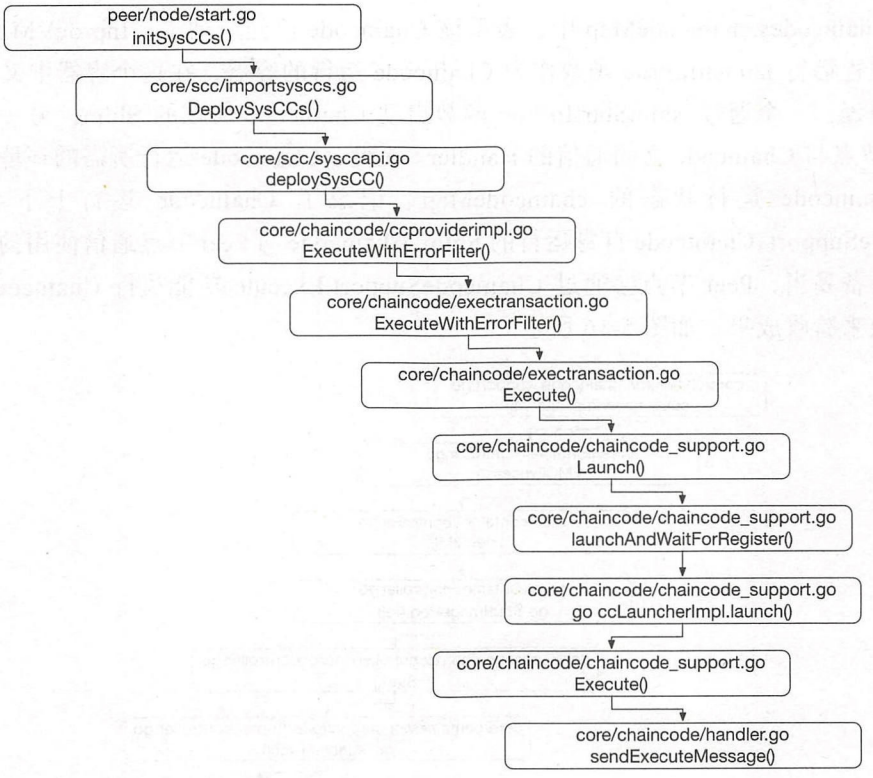


图5-15

启动 Chaincode 的主要代码在 ChaincodeSupport 中，在 ChaincodeSupport.Launch 与 ChaincodeSupport.launchAndWaitForRegister 两个函数中判断 Chaincode 是否启动和使用了锁+双重判断的模式，同时检查 chaincodeHasBeenLaunched 与 launchStarted，即 Chaincode 是否启动或者正在启动，避免 Chaincode 被多重启动。在 launchAndWaitForRegister 中使用了一个 Go 线程调用 ccLauncherImpl.launch 创建 Chaincode 启动进程，然后监听 notfy 与 errChan 通道，判断启动是否成功。

5.3.3 启动 Chaincode

在 ccLauncherImpl.launch 中主要通过 VMCProcess 创建容器，在容器创建成功后，在 InprocVM.Start 函数中调用 preLaunchFunc 函数执行 ChaincodeSupport.preLaunchSetup 函数。preLaunchSetup 函数将启动的 Chaincode 的 handler 注册到 chaincodeSupport.runningChaincodes.chaincodeMap 中，表示该 Chaincode 已经启动了。InprocVM.Start 再启动一个线程运行 launchInProc 函数作为 Chaincode 运行的容器，在这个容器中又分别创建了两个线程：一个通过_shimStartInProc 函数启动 Chaincode 所在的 Shim；另一个建立了在 Peer 节点与 Chaincode 之间通信的 Handler。至此，Chaincode 运行所需的环境，包括记录了 Chaincode 运行状态的 chaincodeMap、记录了 Chaincode 运行上下文环境的 ChaincodeSupport、Chaincode 自身运行的 Shim、Chaincode 与 Peer 节点通信使用的 Handler，都已经准备妥当，Peer 节点会通过 ChaincodeSupport.Execute 功能执行 Chaincode 默认的执行参数来验收成果。如图 5-16 所示。

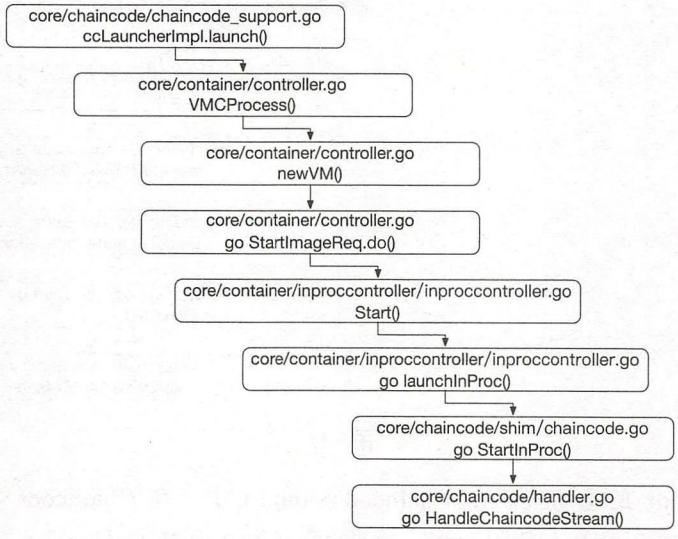


图5-16

5.4 Endorser 的背书流程

从之前的章节中我们了解到，在整个 Fabric 网络的配置中包含了签名验证、MSP、组织 OU、签名策略、隐式元策略，等等，接下来通过分析背书部分的流程来解析这些技术

在代码中的实现。在 Peer 客户端将交易提案打包签名之后，会通过 gRPC 提交给 Peer 节点的 Endorser 部分进行处理，Endorser 的入口在 protos/peer/peer.pb.go 的_Endorser_ProcessProposal_Handler 函数中。

在背书流程中，首先是 core/handlers/auth/filter/expiration.go 中的 validateProposal 对提案的有效性进行验证，简单地检查提案的基础格式是否正确，以及签名证书是否过期。绝大多数流程都在 core/endorser/endorser.go 的 ProcessProposal 中，对于交易提案的详细检查都是在 preProcess 中进行的。

背书的整体流程如图 5-17 所示。

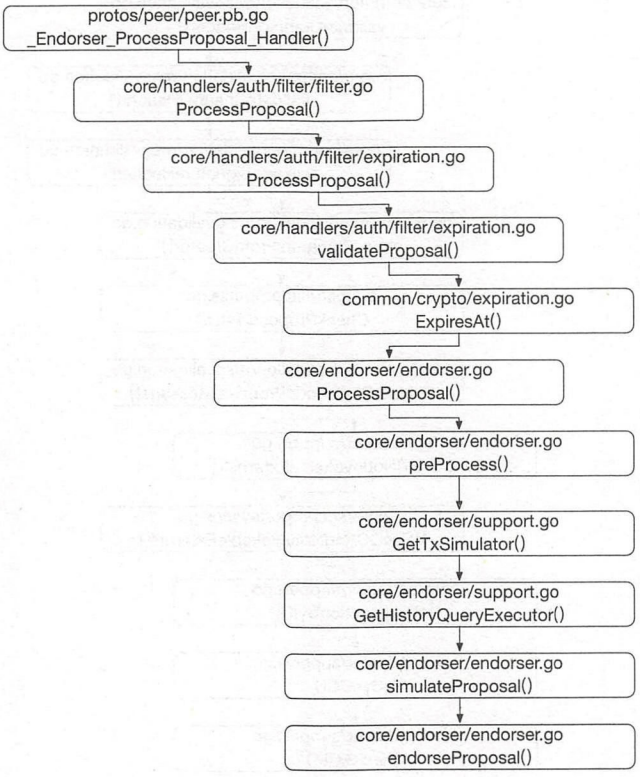


图5-17

5.4.1　preProcess 交易预处理

下面讲解 preProcess 的主要流程。preProcess 主要处理整个提案的有效性检查，首先

通过 ValidateProposalMessage 检查头信息的有效性、签名的有效性、提案 TxID 是否已经被处理及提案消息的有效性。在通过这部分检查后，在 IsSysccAndNotInvokableExternal 中查看调用的 Chaincode 是否是系统 Chaincode，最后通过 CheckACL 进行 ACL 检查。这里从签名验证、TxID 验证、交易策略检查三方面解析 preProcess 的验证流程，如图 5-18 所示。

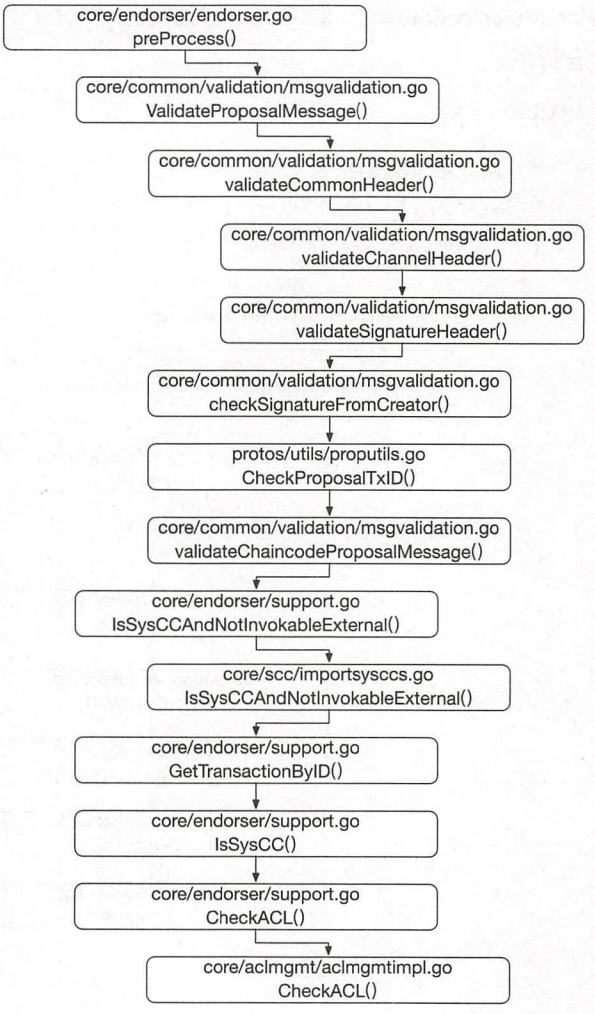


图5-18

5.4.2　checkSignatureFromCreator 检查签名

validateChannelHeader 部分相对简单，我们先来仔细看一下 checkSignatureFromCreator 是如何检查签名的，如图 5-19 所示。

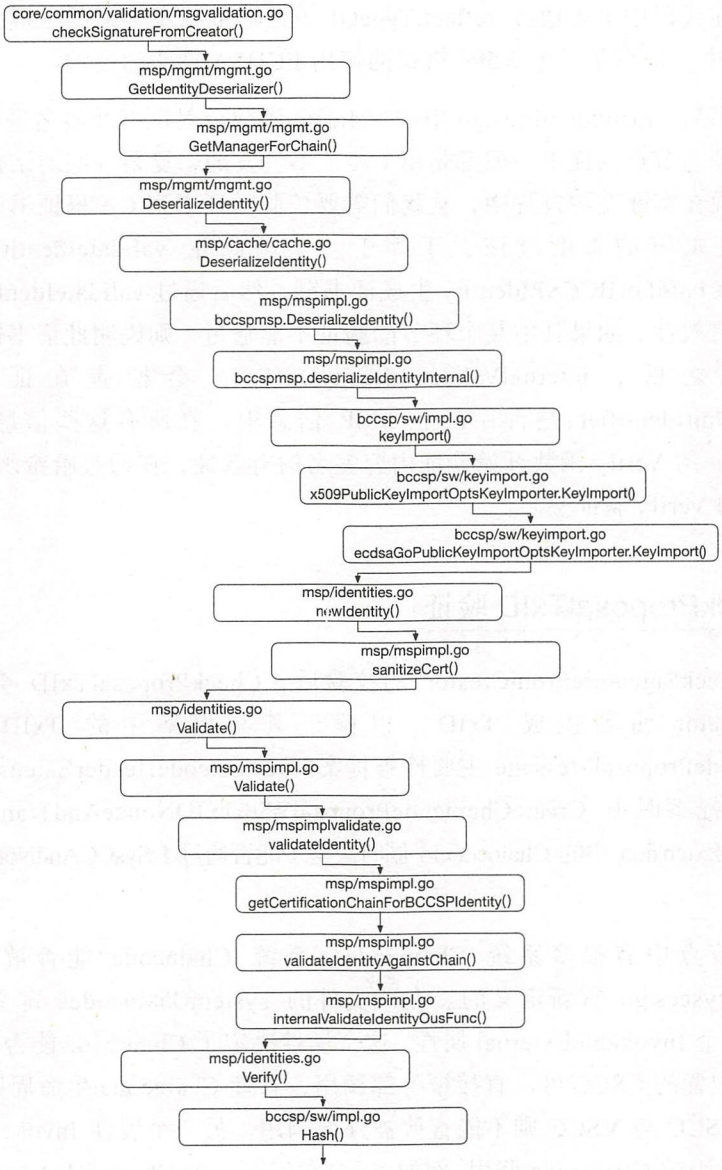


图5-19

我们从之前的内容可以了解到，每个通道都有自己的组织结构，每个组织结构都有自己的 MSP，所以我们在使用 MSP 检查之前，需要通过 GetIdentityDeserializer 获取与当前对应的通道 MSP 相关的信息。在这些信息中包含证书的格式、加密类型等对应的信息，DeserializeIdentity 函数则根据在 MSP 中设置的信息，将在提案中携带的证书信息转换为对应的格式，在代码中主要通过 reflect.TypeOf 方法获取指定类型的处理方法。我们可以看到在图 5-19 中，证书是一个 X509 格式的使用 ECDSA 算法的公钥。

在获取证书后，msp/identities.go 中的 Validate 函数会对证书及签名进行验证。我们知道，Peer 客户端与节点的证书一般都是由 CA 证书签发的，复杂一些的结构则会使用中间证书，也就是说在实际使用过程中，从我们需要校验的证书到 CA 根证书应该形成一条证书链，且此证书链的长度应该大于等于 2。因此在 validateIdentity 中首先通过 getCertificationChainForBCCSPIdentity 生成证书链，然后通过 validateIdentityAgainstChain 验证证书链的有效性，如果其中某个环节的验证不能通过，则说明此证书链无效。在通过证书链验证之后，internalValidateIdentityOusFunc 会检查在证书中声明的 OrganizationalUnitIdentifier 是否存在于 MSP 信息中。在所有这些信息都匹配之后，msp/identities.go 的 Verify 函数开始验证内容签名的有效性，在通过哈希函数生成摘要后，通过 ECDSA 的 Verify 验证签名。

5.4.3 CheckProposalTxID 验证

在完成 checkSignatureFromCreator 的校验后，CheckProposalTxID 会通过提案中的 Nonce 与 Creator 重新生成 TxID，以验证其与提案中的 TxID 是否一致。validateChaincodeProposalMessage 主要检查提案的 ChaincodeHeaderExtension 的值，这些值在构造交易提案时由 CreateChaincodeProposalWithTxIDNonceAndTransient 初始化。ChaincodeHeaderExtension 中的 ChaincodeId 同时决定了能否通过 IsSysCCAndNotInvokableExternal 的检查。

在 Peer 节点中有很多系统 Chaincode，系统 Chaincode 能否被外部调用是由 core/scc/importsysccs.go 重新定义的。在该文件的 systemChaincodes 定义中可以看到，Chaincode 有一个 InvokableExternal 属性，这个属性决定了 Chaincode 能否被外部调用，例如，用于修改配置的 CSCC 可以直接被外部调用，管理 Chaincode 生命周期的 LSCC 也能被外部调用，ESCC 与 VSCC 则不能直接被外部调用。另一个属性 InvokableCC2CC 决定了 Chaincode 能否被 Chaincode 调用，例如，用户在编写一个 Chaincode 时，可以调用 QSCC

对某个 Key 进行查询，但不能调用 ESCC 生成一个背书记录。

接下来，系统需要进一步校验 TxID。我们之前通过重新生成 TxID 的方式来检查提案中的 TxID 与 Nonce、Creator 是否匹配，但这不能阻止恶意的重放攻击，因此需要在已有的账本记录中查询是否存在过这个 TxID，如图 5-20 所示。

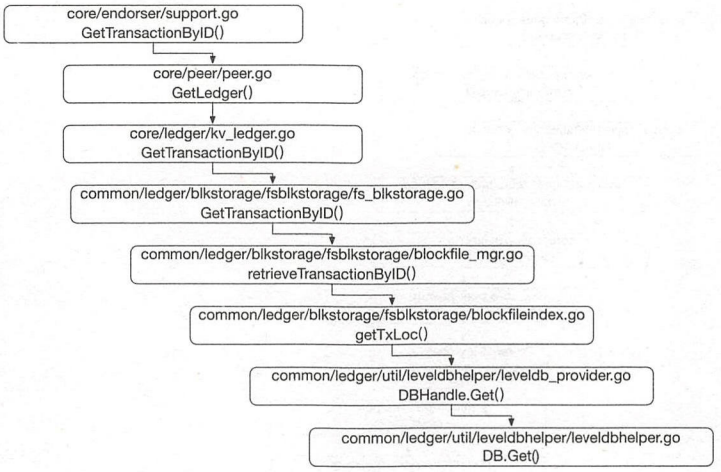


图5-20

查询 TxID 的流程其实非常简单，Peer 节点在根据通道 ID 查询到对应的 kvLedger 之后，会从 kvLedger 的 blockStore 中调取索引数据库 index，通过 index 查询对应的 TxID 的情况。

从安全角度来说，如果这部分只是用于重放攻击的，则处理位置和时机都略有问题。如果指定的交易并未提交（Commit），则在查询 index 时查询不到对应的交易记录，重放请求依然会进入背书流程；即使已经提交了，在面对大量的重放攻击时，之前 SHA256 的哈希运算和签名验证等类似操作都将消耗大量的性能资源。如果 Peer 节点有可能暴露在公网环境下，则这是需要考虑的因素。

5.4.4　策略评估

回头看一下，已经有证书验证、签名验证、交易提案格式验证、TxID 验证了，是不是可以背书了？有没有感觉还差了点什么？对了，在 5.1.4 节着重讲解的策略评估到哪里去了？

在 core/endorser/support.go 的 CheckACL 函数中实现了对交易提案的策略评估，如图 5-21 所示。

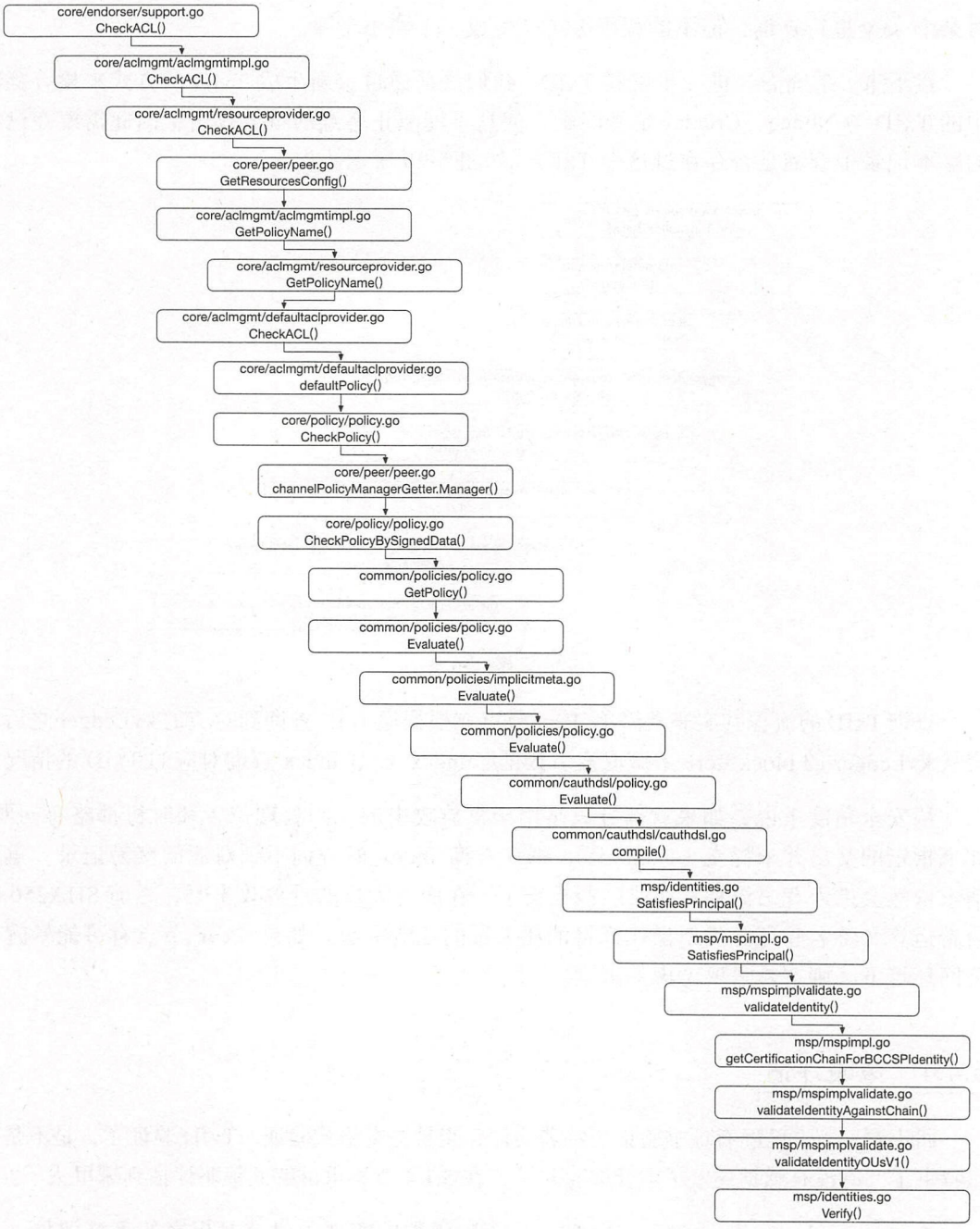

core/endorser/support.go
CheckACL()
core/aclmgmt/aclmgmtimpl.go
CheckACL()
core/aclmgmt/resourceprovider.go
CheckACL()
core/peer/peer.go
GetResourcesConfig()
core/aclmgmt/aclmgmtimpl.go
GetPolicyName()
core/aclmgmt/resourceprovider.go
GetPolicyName()
core/aclmgmt/defaultaclprovider.go
CheckACL()
core/aclmgmt/defaultaclprovider.go
defaultPolicy()
core/policy/policy.go
CheckPolicy()
core/peer/peer.go
channelPolicyManagerGetter.Manager()
core/policy/policy.go
CheckPolicyBySignedData()
common/policies/policy.go
GetPolicy()
common/policies/policy.go
Evaluate()
common/policies/implicitmeta.go
Evaluate()
common/policies/policy.go
Evaluate()
common/cauthdsl/policy.go
Evaluate()
common/cauthdsl/cauthdsl.go
compile()
msp/identities.go
SatisfiesPrincipal()
msp/mspimpl.go
SatisfiesPrincipal()
msp/mspimplvalidate.go
validateIdentity()
msp/mspimpl.go
getCertificationChainForBCCSPIdentity()
msp/mspimplvalidate.go
validateIdentityAgainstChain()
msp/mspimplvalidate.go
validateIdentityOUsV1()
msp/identities.go
Verify()

图5-21

ACL（Access Control List）中的 List 在 Fabric 中就是在创世区块中定义的隐式元策略与签名策略列表，CheckACL 用于检查当前提案的行为策略是否匹配 ACL 中的策略。CheckACL 的流程简单来说是在 resourceProvider.CheckACL 中根据提案行为生成类似于“/Channel/Application/Writers”这样的字串，判断这个策略属于隐式元策略还是签名策略；然后在 common/cauthdsl/cauthdsl.go 的 compile 中根据策略类型对签名进行验证，并收集验证后的结果；最后根据收集的结果与策略中的 NOutOf 进行判断，决定提案是否符合策略。

5.4.5　simulateProposal 模拟交易

在所有校验与策略都符合之后，Endorser 通过 simulateProposal 模拟交易，通过 endorseProposal 生成背书结果。

在模拟交易之前，simulateProposal 先将 Proposal 里的 Payload 打包成 Chaincode InvocationSpec。对于 JavaChaincode 来说，需要通过 disableJavaCCInst 的校验才能运行。

模拟交易的执行步骤如下。

（1）simulateProposal 通过 GetChaincodeDefinition 调用 LSCC 查询指定的 Chaincode 的状态。

（2）通过 ExecuteChaincode 调用指定的 Chaincode 来生成模拟交易结果。

（3）将执行后的结果提交给 endorseProposal，simulateProposal 调用 ESCC 对结果进行背书并返回结果。

执行流程如图 5-22 所示。

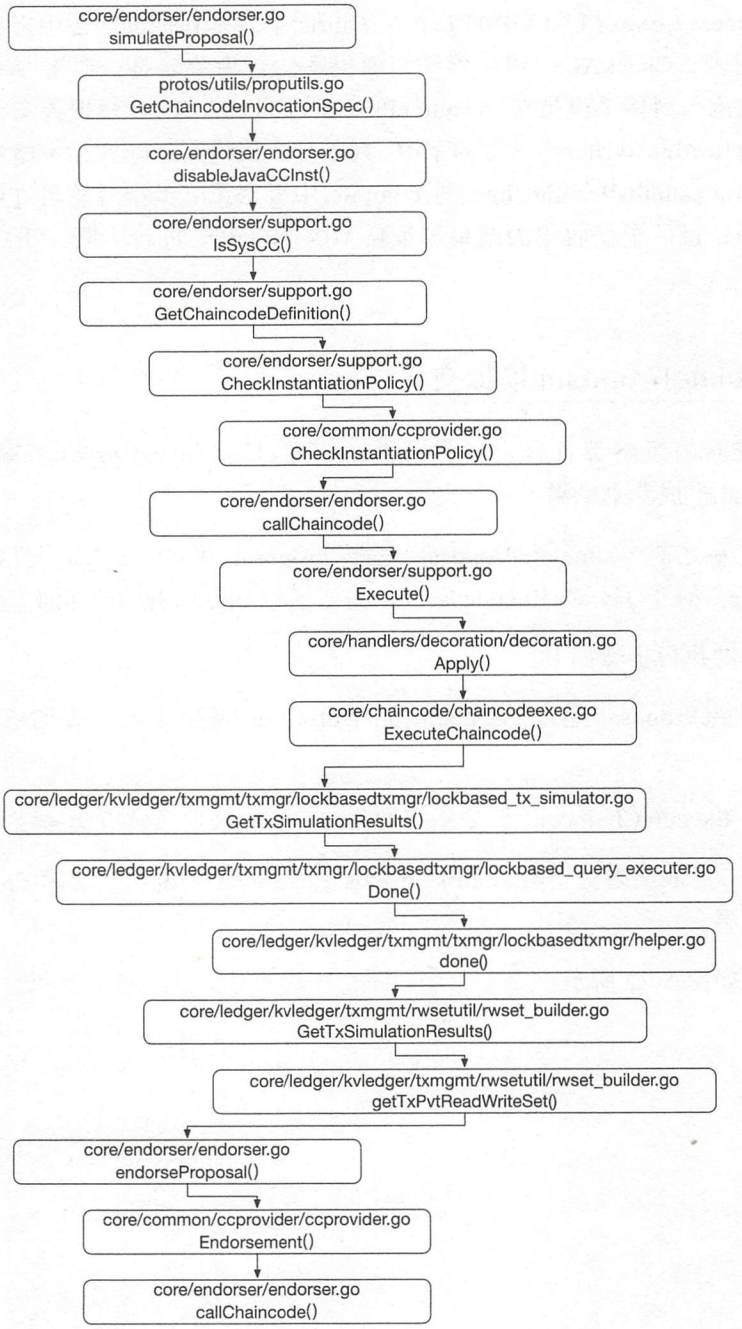

core/endorser/endorser.go
simulateProposal()
protos/utils/proputils.go
GetChaincodeInvocationSpec()
core/endorser/endorser.go
disableJavaCCInst()
core/endorser/support.go
IsSysCC()
core/endorser/support.go
GetChaincodeDefinition()
core/endorser/support.go
CheckInstantiationPolicy()
core/common/ccprovider.go
CheckInstantiationPolicy()
core/endorser/endorser.go
callChaincode()
core/endorser/support.go
Execute()
core/handlers/decoration/decoration.go
Apply()
core/chaincode/chaincodeexec.go
ExecuteChaincode()
core/ledger/kvledger/txmgmt/txmgr/lockbasedtxmgr/lockbased_tx_simulator.go
GetTxSimulationResults()
core/ledger/kvledger/txmgmt/txmgr/lockbasedtxmgr/lockbased_query_executer.go
Done()
core/ledger/kvledger/txmgmt/txmgr/lockbasedtxmgr/helper.go
done()
core/ledger/kvledger/txmgmt/rwsetutil/rwset_builder.go
GetTxSimulationResults()
core/ledger/kvledger/txmgmt/rwsetutil/rwset_builder.go
getTxPvtReadWriteSet()
core/endorser/endorser.go
endorseProposal()
core/common/ccprovider/ccprovider.go
Endorsement()
core/endorser/endorser.go
callChaincode()

图5-22

5.4.6　Chaincode 的调用流程

调用系统 Chaincode 与用户 Chaincode 的流程相同，这里先以 LSCC 的调用为例来解析 Chaincode 的调用流程，如图 5-23 所示。

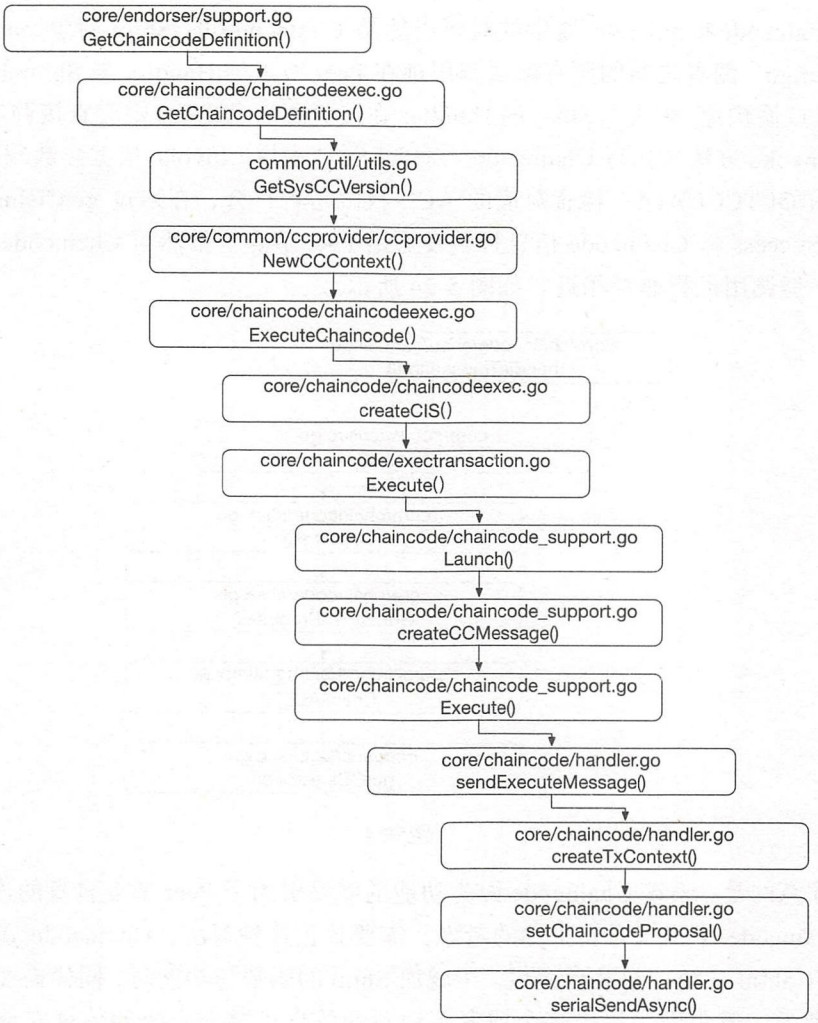


图5-23

执行 Chaincode 的主要 API 都是由 ChaincodeSupport 提供的，每次在 core/chaincode/exectransaction.go 的 Execute 函数中启动 Chaincode 前都会执行 ChaincodeSupport 的 Launch 函数，以确保 Chaincode 在调用时处于运行状态。在准备好调用 Chaincode 所需的参数后，

ChaincodeSupport.Execute 会调用 sendExecuteMessage 将消息提交给 core/chaincode/ handler.go 进行处理，而 handler 自身作为一个状态机，通过 serialSendAsync 启动一个进程来发送消息，在这个发送进程中，通过 handler.ChatStream.Send 发送消息。我们注意到这是一个 ChatStream，那么它是在和谁通信（Chat）呢？

core/chaincode/handler.go 通信的对象当然是 Chaincode 的 Shim 对象 core/chaincode/shim/handler.go。两者之间的所有项目调用都在 Peer 节点的 Handler 与 Shim 的 Handler 之间通过 gRPC 愉快地“聊天”。Shim 的 Handler 在接收到消息之后，则会直接调用 Chaincode 中对应的 Invoke 方法来执行 Chaincode。对于 LSCC 来说，Invoke 根据参数判断 Peer 节点的调用意图 GETCCDATA，检查对应的 ACL 权限是否符合，在通过 getCCInstance 之后，调用 shim.Success 将 Chaincode 信息打包发送给 Peer 节点。虽然与 Chaincode 本身的功能大相径庭，但调用流程非常相近。如图 5-24 所示。

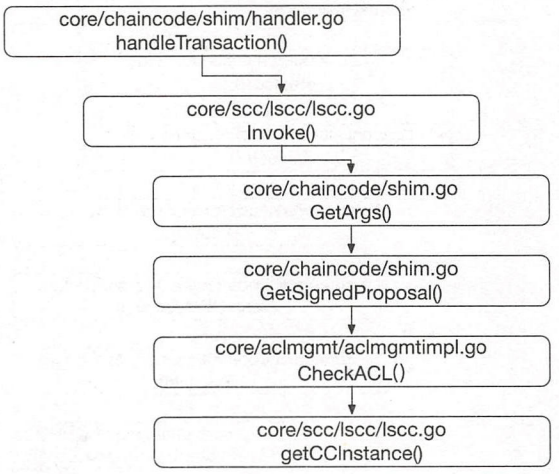


图5-24

需要注意的是，系统 Chaincode 因为功能的需要引用了 Peer 节点自身的许多函数，在第三方 Chaincode 中出于对安全性的考虑，需要禁止这种写法，Chaincode 需要的功能与函数应当由 Shim 在统一封装后提供。在增加 Shim 的函数与功能时，同样需要从全局安全的角度来考虑，避免因为提供多余或者不合适的信息给整个系统的信息安全带来风险。Shim 为 Chaincode 提供的接口主要在 core/chaincode/shim/interfaces_stable.go 中。

5.4.7　RWSet 与防双花攻击

在通过 LSCC 查询到 Chaincode 的运行信息后，系统会执行对应的 Chaincode。在我们的例子中主要使用了 Fabric 源码下的示例代码 example/chaincode/go/chaincode_example02。该示例代码用于处理简单的 Key-Value 读取与存储。在模拟交易调用该 Chaincode 之后，会通过 GetTxSimulationResults 收集模拟交易的结果，将结果放到 RWSetBuilder 的结果中，如果在调试环境下，则可以通过打印 RWSetBuilder 看到如下结果：

```
b.pubRwBuilderMap["mycc"].readMap["a"]
<*github.com/hyperledger/fabric/protos/ledger/rwset/kvrwset.KVRead>
Key:"a"
Version:<*github.com/hyperledger/fabric/protos/ledger/rwset/kvrwset.Version>
BlockNum:3
TxNum:0

b.pubRwBuilderMap["mycc"].writeMap["a"]
<*github.com/hyperledger/fabric/protos/ledger/rwset/kvrwset.KVWrite>
Key:"a"
IsDelete:false
Value:<[]uint8> (length: 2, cap: 2)
[0]:56
[1]:55
```

可以看到，最终生成的结果在 ReadMap 中指定了值的版本，在 WriteMap 中指定了结果。这样做的好处是，在 VSCC 进行 MVCC（Multi-Version Concurrency Control，多版本并发控制）校验时，很容易判断当前值修改的版本，避免双花攻击。双花攻击通常指在同一时间对同一账户的钱进行多次支付，Endorser 虽然会计算出模拟交易结果，但并不确定交易最终是否发生，因此模拟交易结果也不会被写入区块或者 PrivateData 中。因此，Fabric 非常容易遭受双花攻击，而在模拟交易结果中写入版本号是对抗双花攻击的非常好的一种手段。在同一个批次的交易处理中，如果发现有两笔或者两笔以上的交易操作的对象版本发生冲突，则会返回 TxValidationCode_MVCC_READ_CONFLICT 错误，具体的代码可以在 core/ledger/ kvledger/txmgmt/validator/statebasedval/state_based_validator.go 的 Validator.validateTx 中找到。当然，这样做并非万无一失，其缺点可能是由于 Gossip 协议传输慢、网络及区块同步慢等各种因素，导致账户版本同步时间过长而交易失败及重试的次数太多，在实际高并发的应用场景下，这是我们必须考虑的因素。

5.4.8 ESCC 背书流程

在生成模拟交易结果后，系统会调用 ESCC 对结果进行背书，如图 5-25 所示。

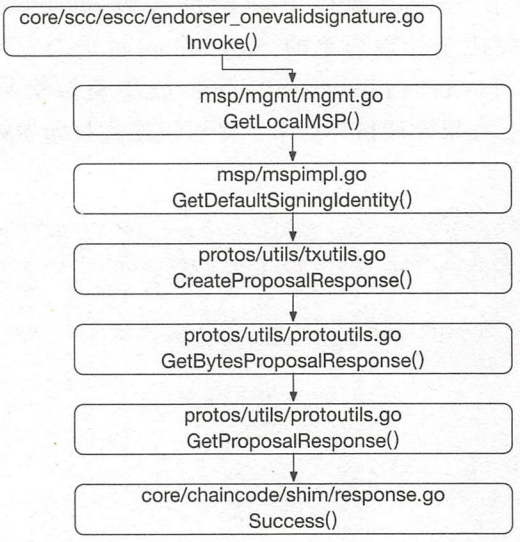


图5-25

通过查看 ESCC 的代码，我们很容易就能发现，ESCC 其实只是对结果的格式合法性进行判断，并没有对值的有效性进行判断，在判断完之后调用本地 MSP 对结果进行打包签名，然后由 Endorser 将打包后的结果返回给客户端。至此，背书流程结束。

本章详细讲解了创世区块、交易提交、交易背书、Chaincode 的启动等内容，深入探讨了 Chaincode 的运行机制、策略的定义与实现，以及 Fabric 对于重放攻击及双花攻击的防范与设置。由于篇幅的限制，这里只是大致浏览了一遍代码流程与部分细节，远不能覆盖 Fabric 的所有环节。希望这些内容能够让我们对 Fabric 的架构有更清晰的认识，并且举一反三地分析 Fabric 中的代码与设计。

第6章
深入理解Fabric的安全机制

6.1 Fabric安全概述

作为一个企业级区块链产品，Fabric区块链平台相对于以太坊这种公有链来说，在安全方面做了很多强化措施，主要有以下几方面。

- 成员管理服务：确保只有那些“认识的”“靠谱的”小伙伴（企业组织）加入，确保网络成员“苗红根正”。
- 智能合约安全机制：作为区块链交易承载主体的智能合约，其安全性极其重要，Fabric平台从智能合约的源代码、发布到执行环境等各个方面都做了安全防护。
- 交易安全与隐私保护：由于Fabric区块链是一个企业级联盟链，因此在交易过程中存在多个交易主体，做好交易安全与交易内容隐私保护是关键的安全特性之一。

6.1.1 成员管理服务

Fabric是一个联盟区块链，不是一个公有链，它的目标是建立以企业为核心的可信计算环境，以确保商业安全。为此，在Fabric平台架构中专门设计了MSP（Membership Service Provider）来实现Fabric网络中来自不同组织机构的成员的身份识别与管理。Fabric的成员管理服务是Fabric安全机制的基石与核心，也是Fabric到目前为止最贴近联盟链的区块

链设计思想之一。由于联盟链需要考虑商业应用对安全、隐私、监管、审计等的需求，因此，成员必须“被许可”，才能加入 Fabric 网络。Fabric 成员管理服务为整个区块链网络提供了身份管理、隐私、保密和可审计的服务，通过公钥基础设施 PKI 使非许可制的区块链变成许可制的区块链。

（1）成员身份管理的第 1 个目标是解决成员的核实与准入问题。

这个目标看起来很简单，实则不然。这是因为 Fabric 网络中的一个“成员”对应的是一个“商业实体组织”，确切地说，通常是一个大型商业机构，比如银行、证券、保险等。这些大型机构在组织结构上是有上下级关系的，同时，在地理位置上是分布式的，因此，这样的商业机构在作为一个成员加入 Fabric 网络中时，其背后其实是分散在多处的几个内部组织加入并对应多个 Peer 节点。由于一个商业机构的下属分支也是有自己的独立身份的，并且作为一个独立的执行机构去从事商业活动，因此，Fabric 需要识别每个下属分支的身份。于是，我们看到在正常情况下，一个商业机构（Org1）对应一个 MSP 实例，同时加入 Fabric 网络中的每个下属分支都需要有自己的独立 CA 证书，并且对应不同的 Peer 实例。此外，每个 MSP 实例都有一个独立的 CA 根证书，这个根证书用来对其服务范围内的所有 Peer 节点的证书进行签名。这样一来，Fabric 的交易网络就确保了所有参与者都拥有“组织鉴定和核实”的“已知身份”，每个参与者所属的商业机构负责该组织内所有 Peer 节点的证书发布与身份核实工作，这种两级的成员身份管理方式具有很好的灵活性，非常适合企业级联盟链的定位。

（2）成员管理服务的第 2 个目标是实现交易的问责与审计。

问责制意味着系统中的任何成员都需要为自己的行为负责，并且不可抵赖，健全的问责制可确保诚实的用户不会因为其他用户的违规交易而被指控或牵连。问责与审计通常是密切关联在一起的，这也属于 B2B 系统的核心安全需求之一，成员管理服务恰恰是保障问责制实现的关键。在 Fabric 中，每个事务都包含事务发起方成员的数字签名，因为数字签名具有不可伪造的特点，所以如果在 Fabric 网络中有某类特定交易属于异常交易或者违规交易，Fabric 就可以迅速定位并追溯到此交易的发起人，并以此为证，对相关责任方进行问责。

在区块链、公有链中，每个参与者都能够获得完整的数据备份，所有交易数据都是公开和透明的，这是区块链的优势；但另一方面，对于很多区块链应用方来说，这个特点又是致命的，因为在很多时候，不仅用户本身希望自己的账户隐私和交易信息被保护，而且对于商业机构来说，交易数据更是这些机构的重要资产和商业机密，他们不希望这些数据

被同行看见，比如 Fabric 网络上的两家相互竞争的企业都拒绝让对方看到自己的交易数据，这就需要解决交易中的隐私问题。因此，Fabric 既要实现交易的问责制和不可抵赖性，又要保护参与者的商业机密及成员的交易隐私。

6.1.2　交易安全与隐私保护

在 Fabric 中发生的交易可以分为以下两类。

- 第 1 类：普通交易（None-confidential Transaction），也可以将其理解为公开交易，即原始交易数据是以明文方式存储到账本中的，所有能访问这个账本的成员都可以看到交易信息。
- 第 2 类：秘密交易（Confidential Transaction），也可以理解为匿名交易，即交易的原始数据被存储到账本中的时候，发起者的身份信息被“抹去”，交易内容被加密存储，只有交易发起者、平台监管者及审计者才能解密交易内容。

“秘密交易”的实现很巧妙，既保护了用户的交易隐私，又实现了交易的问责与审计要求，下面分析它的实现细节。

在 Fabric 交易过程中涉及的两类主要证书分别是 E-cert（Enrollment-cert）和 T-cert（Transaction-cert），前者是一个长久有效的证书，是注册证书颁发机构（ECA）颁发给 Fabric 成员的证书，也就是我们常说的成员证书；后者则是一个临时证书（Short-term Cert），是交易认证中心（TCA）颁发给 Fabric 成员用于每个“秘密交易”的短期证书。T-cert 可以安全地给一个交易授权，并且配置成不携带用户的身份信息，而 T-cert 证书与其持有者的关联关系只有交易认证中心与审计者知道，他人无法获知，这使得用户能以匿名方式参与到系统中。尽管一个 T-cert 证书可用于多个事务，但是为了保护隐私，还是建议每个事务都用一个新的事务证书，这样一来，同一用户的多个交易不能被关联起来，这阻止了他人去发掘交易之间的关联性，从而有效保护用户的活动（交易）隐私。

除了交易的签名，在 Fabric 中，交易的安全性还涉及交易数据传输的安全问题，因为交易数据从客户端发出，最终被写入各个 Committer 节点，中途经历了很多网络传输节点。为了保证交易数据的真实性，Fabric 开启 TLS 安全传输通道来传输数据，这样一来，Fabric 网络中的所有节点都开启了安全传输通道，避免了交易数据在传输过程中被篡改及被偷窥的风险。Fabric 专门提供了第三种证书——TLS-cert，来保证 Fabric 网络中所有节点之间的通信信道的安全性。TLS-cert 证书由 TLSCA 颁发，也可以由 E-Cert 代替，但这样做削弱了安全性。在 IBM Blockchain Platform 中，Fabric TLS 的传输特性默认是启动状态的。

总结一下，在 Fabric 中主要通过三类证书的配合使用来保证交易的安全性、可审计性及隐私性，其中，E-cert用于识别与鉴定交易双方的身份，T-cert用于保护交易隐私，TLS-cert则用于解决 Fabric 网络中交易数据的安全传输问题。

6.1.3 智能合约的安全机制

智能合约是 Fabric 用户开发的一段实现了某个商务合约的可执行程序，这段代码被嵌入 Fabric 平台中执行，并且涉及 Fabric 区块链账本数据的读写，因此，智能合约的安全性在整个区块链平台中显得尤为重要。Fabric 在智能合约的安全性方面也是精心设计的。

为了避免智能合约的程序代码对 Fabric 平台的安全性造成威胁，Fabric 采取了进程间隔离的技术将智能合约代码的进程隔离到系统之外，并通过 Socket 通信方式远程调用智能合约，这样做的好处很明显，如下所述。

（1）智能合约在执行过程中的性能问题不会连累平台，因为 Socket 远程调用可以采用异步非阻塞的模式，避免 Fabric 平台的线程受到智能合约执行性能的影响。

（2）智能合约进程无法访问 Fabric 进程的内存，无法侵入或对平台带来安全隐患。

但传统的单进程模式的智能合约仍然存在一些问题，这些问题如下。

（1）安装部署比较麻烦，不同的 Linux 发行版可能需要不同的部署方式。

（2）仍然与 Fabric 进程共享主机文件系统，因此还是可以直接读写 Fabric 的账本数据文件和系统配置文件等，给 Fabric 平台带来安全隐患。

（3）智能合约进程的内存与 CPU 消耗无法被限制，一个恶意的智能合约程序可能导致系统资源被消耗殆尽，影响 Fabric 平台的稳定性。

为此，Fabric 采用了先进的容器技术，将用户开发的智能合约代码打包为 Docker 镜像，然后以容器方式启动，从而优雅地解决了上述问题。需要说明的是，打包智能合约代码生成 Docker 镜像的过程是 Fabric 自动完成的，当我们初始化（instantiate）一个智能合约时，如果此智能合约的 Docker 镜像还不存在，则系统会执行自动打包逻辑。因此，智能合约是一段运行在独立的容器内的程序，它不能与外部有 I/O 通信，不能直接操作账本，只能与 Endorser 之间实现被动的 gRPC 通信，与 Fabric 系统及账本数据完全隔离。

智能合约是一种高度可靠的多方信任的、无法抵赖、无法篡改的“商业合约”，为了保证这一点，智能合约在代码开发完毕后，需要通过 package 命令打包为一个可部署的安

装包（package），该安装包需要遵循 Chaincode Deployment Spec（CDS）规范。CDS 规范中最重要的一组信息是智能合约的拥有者的共同签名，这些签名的作用如下。

（1）明确智能合约的所有权。

（2）用来检测和验证智能合约的真实性。

（3）用来验证智能合约代码是否被篡改。

这样一来，Fabric 中的智能合约就类似签名过的 Java Applet，一旦被部署到 Fabric 网络中生效，就如同纸面合同一样无法被篡改了，直到下一次升级版本后重新部署，这样 Fabric 便真正成为一个多方信任的区块链账本平台了。

最后，我们分析一下智能合约中的另一个安全问题，即如何保证 Fabric 账本数据的安全性？

答案很简单，Fabric 不允许在智能合约代码中直接修改区块链账本数据。如图 6-1 所示，智能合约在执行过程中并不能直接修改区块链的账本数据，而是在执行完毕后返回一个 ReadWriteSet 对象，其中包括智能合约操作过的 Key-Value 集合，Write Set 是需要写入账本中的数据集。

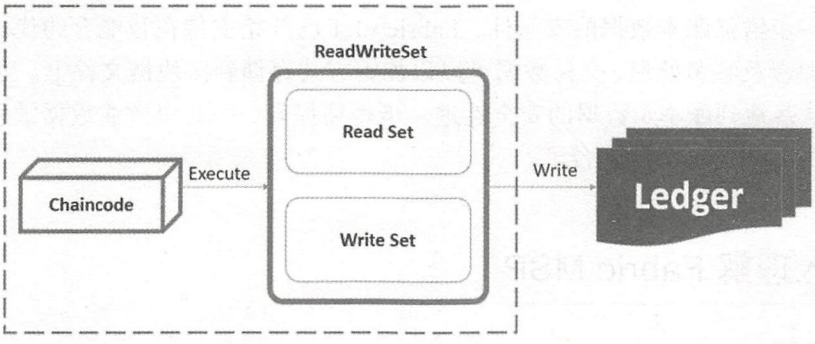


图6-1

以之前的 example02 中的转账方法 invoke 为例，下面这段代码在调用完成后产生的 Write Set 集合中有 A、B 两个 Key-Value 键值对：

```
err = stub.PutState(A, []byte(strconv.Itoa(Aval)))
err = stub.PutState(B, []byte(strconv.Itoa(Bval)))
```

在 invoke 转账交易提案被 Endorser 背书通过以后，上述 Write Set 集合才真正生效，

最终被Committer节点写入区块链账本中永久保存。实际上，Committer还会对ReadWriteSet的数据进行版本检查，确保在事务操作过程中不会产生脏数据，可见 Fabric 对账本数据安全问题的防范有多严谨。如图 6-2 所示为 Write Set 从产生到写入账本的完整流程。

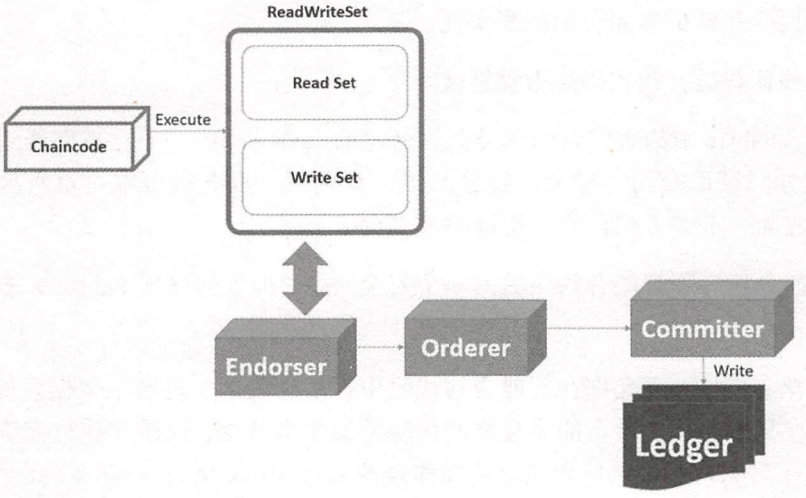


图6-2

为了进一步增强账本数据的安全性，Fabric v1.1 已开始支持在智能合约代码级别对交易数据进行加密及签名处理，交易数据可以以加密方式存储到区块链文件上。至此，Fabric 交易（包括从落地到账本）数据的安全性进一步得到提升，一些对商业数据敏感的应用就可以安心使用 Fabric 区块链平台了。

6.2 深入理解 Fabric MSP

6.2.1 MSP 模型

Fabric 中的成员管理服务主要是通过 MSP 来实现的，MSP 有以下两重含义。

◎ 首先，它是一个以 Fabric 中的成员（Member）的身份认证功能为核心的抽象组件（模型），涵盖了成员身份识别、证书颁发、成员准入、签名加密和证书注销等主要服务。

◎ 其次，MSP 在 Fabric 里又是一个具体实现，它采用了 PKI 公钥基础设施与数字证书技术来实现 Fabric 成员身份管理功能。在最简单的 Fabric 网络中，通常至少有两个 MSP 实例，一个供 Orderer 节点使用，一个供 Peer 节点使用。

每个 MSP 实例都需要指定一个全局唯一的名称，也被称为 MSP 标识符或 MSP ID。我们通常可以用一个组织（Org）的名称来命名它，因为在大多数情况下（也是 Fabric 官方建议的方式），我们会为每个组织建立一个 MSP 实例，并一一对应起来，比如我们会命名 Org1 对应的 MSP 为 Org1MSP。

那么 MSP 实例里的成员是什么，又是如何表示的呢？

MSP 里的每个成员都被称为一个 Identity，Identity 表现为一个 X509 的证书（及对应的私钥）。在 Fabric 中 MSP 的默认实现如下。

◎ 每个 MSP 实例都拥有一个自签名的 CA 根证书（MSP CA Root Cert），用来对 MSP 成员的证书（Enrollment-cert）进行签名并完成对成员身份的验证。
◎ 每个 MSP 实例都拥有一个自签名的用于 TLS 传输的 CA 根证书（MSP TLS CA Root Cert）。
◎ 需要在不同的文件夹下存放 MSP CA Root Cert 和 MSP TLS CA Root Cert。为了安全起见，Fabric 强烈建议在配置 MSP 的过程中不共用同一个证书文件。
◎ 每个 MSP 成员（组织、Fabric 节点、个人、客户端程序）的 X509 证书都必须被根证书签名。
◎ MSP 成员的 X509 证书中的 OU 字段给出了其从属的组织（Org）。
◎ 每个 MSP 实例都可以拥有一个或多个管理员，管理员可以修改此 MSP 实例的配置。

每个 MSP 成员的 X509 证书实际上就是 E-cert 证书，我们在增加一个 MSP 成员（Enroll an Identity）时，只要生成一个新的签名证书即可，为此，我们可以使用 OpenSSL 工具。但 Fabric 不支持 RSA 密钥的证书，目前只支持椭圆曲线加密算法（Elliptic Curve Cryptography，ECC）的证书，因此我们可以使用 Fabric 提供的 crytogen 工具来生成符合要求的 MSP 证书。此外，Fabric CA 组件也可被用于生成 MSP 所需的密钥和证书，后面会详细说明。

MSP 成员的角色（MSP Role）共有 4 种，分别是 Member、Admin、Peer 及 Client，对应的源码如下：

```
message MSPRole {
    string msp_identifier = 1;
    enum MSPRoleType {
        MEMBER = 0;  // Represents an MSP Member
        ADMIN  = 1;  // Represents an MSP Admin
        CLIENT = 2; // Represents an MSP Client
        PEER = 3;   // Represents an MSP Peer
    }
    MSPRoleType role = 2;
}
```

这里比较容易理解的是 Admin、Peer、Client 这三种角色。其中，Admin 为此 MSP 的管理者，管理者可以修改 MSP 的配置信息；Peer 则说明此证书（成员）对应为某个 Fabric Peer 节点；Client 说明此成员为某个 Fabric 客户端的程序。那么如何理解 Member 角色？看 Fabric 中的一段源码就明白了：

```
// Principal contains the msp role
      mspRole := &m.MSPRole{}
       …
      // now we validate the different msp roles
      switch mspRole.Role {
      case m.MSPRole_MEMBER:
          // in the case of member, we simply check
          // whether this identity is valid for the MSP
          mspLogger.Debugf("Checking if identity satisfies MEMBER role for %s",
msp.name)
          return msp.Validate(id)
      case m.MSPRole_ADMIN:
          …
          }
```

我们可以理解上述源码为，一个 MSP 实例中的任一合法成员都拥有 Member 角色，比如，我们可能在 Fabric 的背书策略规则中见到以下配置：

```
OR('Org1.member', 'Org2.member')
```

这里的 Org1、Org2 其实就是 MSP ID，上述配置表示请求 Org1 或者 Org2 这两个 MSP 实例中的任一成员对交易提案进行背书签名即可，也意味着 Client 类型的成员也能背书签名！在正常情况下，我们当然只希望 Peer 节点对交易提案进行背书，因此我们可以把背书策略的规则改写如下：

```
OR('Org1.peer, 'Org2. Peer')
```

当一个 Peer 节点第一次连接到通道的时候，该 Peer 节点的 MSP 对象负责完成成员身份的认证过程，随后在会话（Session）中完成成员身份绑定的逻辑，Fabric 网络中的点对点（P2P）消息的安全性由 Peer 的 TLS 层来处理，除了特殊场合，这些消息不需要再用节点的 E-Cert 进行签名。

6.2.2　MSP 的证书体系

MSP 的 CA 根证书为自签名证书，CA 根证书可以形成一棵根证书链，如下所示。

```
    rCA1
      /   \
    iCA1   iCA2
     / \     |
iCA11 iCA12 id
```

MSP 为每个成员颁发的证书都是永久证书，那么一旦这个成员被清理出 Fabric 网络，又该如何收回其合法身份（吊销 CA 证书）呢？CA 证书的吊销存在两种机制：第 1 种机制是在线检查，Client 端向 CA 机构发送请求，检查 CA 证书是否有效；第 2 种机制是在 Client 端存储一份由 CA 机构提供的证书吊销列表 CRL（Certificate Revocation List）。第 1 种机制要求 CA 机构的查询服务器具备良好的性能，建议对安全性要求很高的网站采用这种机制。在 Fabric MSP 中采用的是第 2 种机制，即只要把该成员的证书放入 CRL 中即可收回授权。

MSP 可以配置一组根证书（rCA，全称为 root Certificate Authorities，指根管理证书）权限，并且可以选择一组中间证书授权机构（iCAs，全称为 intermediate Certificate Authorities）。一个 MSP 的 iCA 证书必须有 MSP 中任一 rCA 或 iCA 的签名。MSP 的配置可能包含证书撤销列表或 CRL，如果任一 MSP 的 rCA 都被列入 CRL 中，在 MSP 配置的 CRL 中就不能包含 iCA，否则 MSP 的设置将会失效。

接下来我们看看一个 MSP 实例的证书目录结构，如下所述。

- cacerts：PEM 格式的 MSP 的根 CA 证书目录。
- admincerts：MSP 的管理员（角色）证书目录。
- keystore：节点的私钥目录。
- signcerts：节点 Identity 的证书，其实就是节点的 CA 证书。

下面是一个真实的 MSP 的证书目录结构：

```
/root/fabric/
└── msp
    ├── admincerts
    │   └── cert.pem
    ├── cacerts
    │   └── localhost-7054.pem
    ├── keystore
    │   └── a044e43ad1fd7cdfd1fd995abaef53895534bd70e8cdfdb665430d12665f2041_sk
    └── signcerts
        └── cert.pem
```

在 Fabric Orderer 节点的进程启动参数中，下面两个参数用来指定该节点的 MSP 信息：

```
ORDERER_GENERAL_LOCALMSPID=OrdererMSP;
ORDERER_GENERAL_LOCALMSPDIR=/shared/crypto-config/…/msp。
```

在 Peer 节点进程的启动参数中也有类似的两个参数：

- CORE_PEER_LOCALMSPID=Org1MSP；
- CORE_PEER_MSPCONFIGPATH=/shared/.../msp/。

在 Fabric 自带的例子中，通过命令行工具 cryptogen 生成的 org1.example.com 的 MSP 文件的目录如下：

```
org1.example.com/
├── ca     # 存放组织Org1的根证书和对应的私钥文件
│   ├── ca.org1.example.com-cert.pem
│   └── xxxxx_sk
├── msp    # 存放该组织的身份信息
│       ├── admincerts          # 组织管理员的身份验证证书
│       │       └── Admin@org1.example.com-cert.pem
│       ├── cacerts # 组织的根证书，与CA目录下的文件相同
│       │       └── ca.org1.example.com-cert.pem
│       └── tlscacerts          # 用于TLS的CA证书，自签名
│               └── tlsca.org1.example.com-cert.pem
├── peers   # 存放属于该组织的所有Peer节点
│       ├── peer0.org1.example.com    # 第1个Peer节点的信息
│       └── peer1.org1.example.com    # 第2个Peer节点的信息
│           ......................
├── tlsca    # 存放与TLS相关的证书和私钥
└── users    # 存放属于该组织的用户
        ├── Admin@org1.example.com
        ├── Admin@org1.example.com
```

我们从上面的目录结构看到，在一个组织机构中，Fabric 节点（Peer 或者 Orderer）的 MSP（证书）是以一个单独的目录存在的，与组织的其他成员目录（users 目录）是分开存放的。下面是 Peer 节点 peer0.org1.example.com 的 MSP 目录信息：

```
— peer0.org1.example.com
  ├── msp # 与MSP相关的证书
  │   ├── admincerts  # 组织管理员的身份验证证书
  │   │     └── Admin@org1.example.com-cert.pem
  │   ├── cacerts     # 存放组织的根证书
  │   │     └── ca.org1.example.com-cert.pem
  │   ├── keystore    # 本节点的身份私钥，用来签名
  │   │     └── xxxxx_sk
  │   ├── signcerts   # 验证本节点签名的证书，被组织根证书签名
  │   │     └── peer0.org1.example.com-cert.pem
  │   └── tlscacerts  # TLS连接用到的根证书，即组织的TLS根证书
  │         └── tlsca.org1.example.com-cert.pem
  └── tls #与TLS相关的证书
        ├── ca.crt      # 组织的根证书
        ├── server.crt  # 验证本节点签名的证书，被组织根证书签名
        └── server.key  # 本节点的身份私钥，用来签名
```

如果仔细对比 peer0 的 MSP 目录与 Org1 的 MSP 目录，则会发现两者在结构上是完全一致的，peer0 中的 admincerts 与 cacerts 目录是从 Org1 的 MSP 目录里复制而来的，这样设计主要是为了部署方便，因为在 peer0 的 MSP 目录中包含了 Org1 的 CA 证书，就不用再从其他地方获取了。组织的管理员证书也被放入 Peer 的 MSP 目录中，Peer 节点很容易知道某个链接的对端是否是管理员。此外，peer0 的 TLS 证书目录存放了本节点的 TLS-cert 证书，用来保证 Fabric 网络中所有节点之间通信信道的安全性，但这个证书仅用于 TLS 安全传输加密，因此被存放在单独的 TLS 目录里。

下面是 Orderer 节点的 MSP 目录结构，可以看出，它的证书目录结构与 Peer 节点完全一致：

```
├── msp
│   ├── admincerts
│   │     └── Admin@example.com-cert.pem
│   ├── cacerts
│   │     └── ca.example.com-cert.pem
│   ├── keystore
│   │     └── key.pem
│   ├── signcerts
```

```
|   |   └── orderer.example.com-cert.pem
|   └── tlscacerts
|       └── tlsca.example.com-cert.pem
└── tls
    ├── ca.crt
    ├── server.crt
    └── server.key
```

我们继续看 Org1 的 users 目录，以管理员用户为例，会发现它的证书目录结构与 Peer 节点也完全一致：

```
Admin@org1.example.com    # 管理员用户的信息
├── msp # 与MSP相关的证书
│   ├── admincerts    # 组织根证书作为管理员身份验证证书
│   │   └── Admin@org1.example.com-cert.pem
│   ├── cacerts       # 存放组织的根证书
│   │   └── ca.org1.example.com-cert.pem
│   ├── keystore      # 本用户的身份私钥，用来签名
│   │   └── xxxxxxx_sk
│   ├── signcerts     # 管理员用户的身份验证证书
│   │   └── Admin@org1.example.com-cert.pem
│   └── tlscacerts    # TLS连接用的身份证书，即组织TLS证书
│       └── tlsca.org1.example.com-cert.pem
└── tls # 存放与TLS相关的证书和私钥
    ├── ca.crt       # 组织的根证书
    ├── server.crt   # 管理员的用户身份验证证书
    └── server.key   # 管理员的身份私钥，被组织根证书签名
```

6.2.3 MSP 的映射问题

同一个 MSP 实例下的所有成员都可以被视作同一个 Org，拥有一套独立的监管体系，在属于同一个 MSP 实例的不同 Peer 节点之间可以相互分享资源，但一个 Peer 节点只能被单一的 MSP 实例拥有，因此，在真实的环境下如何针对复杂的组织建模对应的 MSP 关系，则成为一个不大不小的问题。

首先，Fabric 官方建议不同的组织对应不同的 MSP 实例，即在组织与 MSP 实例之间一对一映射。比如，我们有 Org1、Org2 这两个独立的组织，则 Fabric 默认会为我们生成如图 6-3 所示的映射关系。

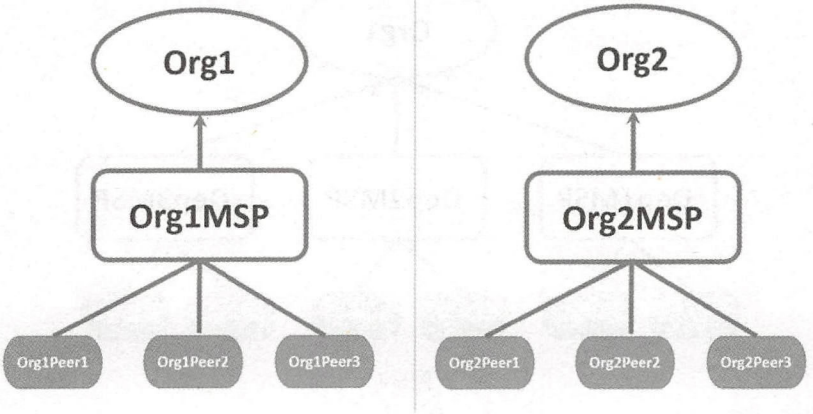


图6-3

问题来了：在不同的 MSP 实例之间是不共享资源的，同一个通道上的 Org1 与 Org2 却可以交易数据。这是因为 Gossip 协议是根据消息中的一个属性来决定如何传播数据的，如下所述。

- UNDEFINED：未定义。
- ORG_ONLY：只在同一组织内传播。
- CHAN_ONLY：只在同一通道内传播。
- CHAN_AND_ORG：同时在同一通道和同一组织内传播。
- CHAN_OR_ORG：既可以在同一通道内传播，也可以在同一组织内传播。

从上面的传播路径来看，的确存在一些消息被限制在同一组织内传播，这样就保证了组织间的隐私，比如财务、销售及研发部门通常有各种隐私，不希望被别的部门知道。如果是出于对事业部独立管理或者组织间隐私方面的考虑，则我们可以把一个组织中的不同部门划分到不同的 MSP 实例中，即组织与 MSP 形成 1 对 n 的映射关系，如图 6-4 所示，这类似于将一个庞大的组织拆分为多个相互独立的组织，比如三国分立。

最后一种比较常见的映射关系是针对“加盟店”这种特殊的组织架构的，即多个独立的组织统一运营，此时，我们可以认为它们在逻辑上是属于同一个组织的，因此可以将它们归到同一个 MSP 实例中管理，如图 6-5 所示。

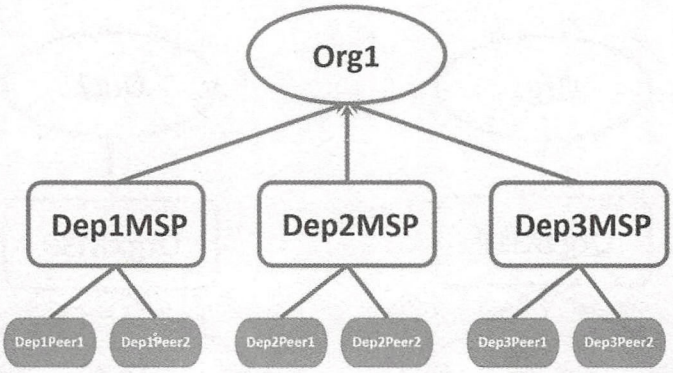


图6-4

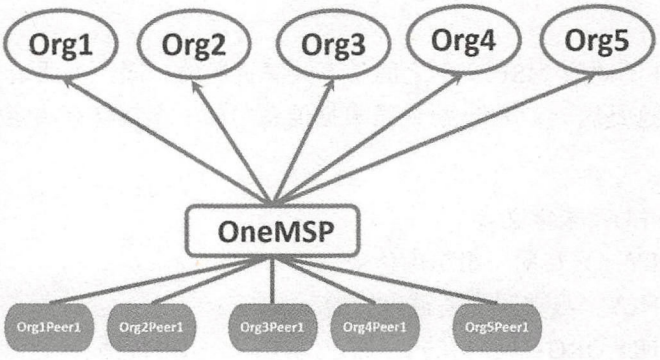


图6-5

6.3 深入理解 Fabric CA

Fabric 提供了两种构建区块链网络的模式，即离线模式和在线模式。在离线模式下，证书通过一个 CA 生成，并分发到所有节点，Peer 节点和 Orderer 节点只能通过离线模式注册。为了客户端可以拉取证书，Fabric CA 提供了在线模式为客户端生成证书。

6.3.1 Fabric CA 架构的组成

Fabric CA 实现了一套完整的、独立的 PKI 公钥基础设施，可以用来生成在 MSP 中所需的全部证书，其主要功能是对 MSP 中的成员身份证书（E-Cert）进行管理，包括生成、

签名、撤销等功能。需要注意的是，Fabric CA 与 Fabric Network 是完全独立的，Fabric CA 可以用于签发 MSP 成员的 E-Cert 证书来配置 MSP，一旦证书被部署，该成员（节点）便可访问 Fabric 网络中的各项资源，而在后续的访问过程中，这些 MSP 成员无须再次向 Fabric CA 请求证书，因此，Fabric CA 并不会造成系统的性能瓶颈。

Fabric CA 的架构组成如图 6-6 所示，它属于典型的 CS（Client and Server）架构，其组成包括两部分：一部分是 Fabric CA Server，它是一个提供 REST API 接口的 Server，默认监听在 7054 端口，CA Server 也是 Fabric CA 功能的核心；另一部分是 Fabric CA 客户端，客户端包括命令行工具及 Fabric CA SDK 开发包，CA 客户端与 Fabric CA Server 的所有通信都是通过 REST API 进行的，前者实现了以命令行方式操作 CA 证书，后者则实现了以编程方式操作 CA 证书，通过客户端生成的 CA 证书在后面可以成为 Peer 节点的 MSP 证书。

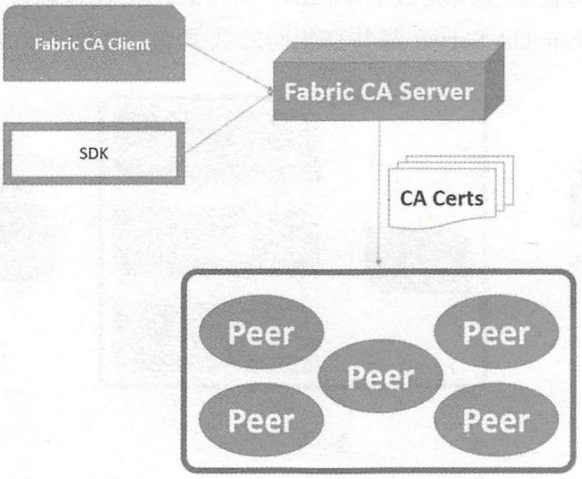


图6-6

Fabric CA Server 可以把生成的证书的属性信息写入关系数据库或者 LDAP 目录服务器中保存下来，所以 Fabric CA Server 的完整架构如图 6-7 所示。Fabric CA Server 默认使用的关系数据库是 SQLite，默认的数据库文件位于服务端程序所在目录下的 fabric-ca-server.db 中。

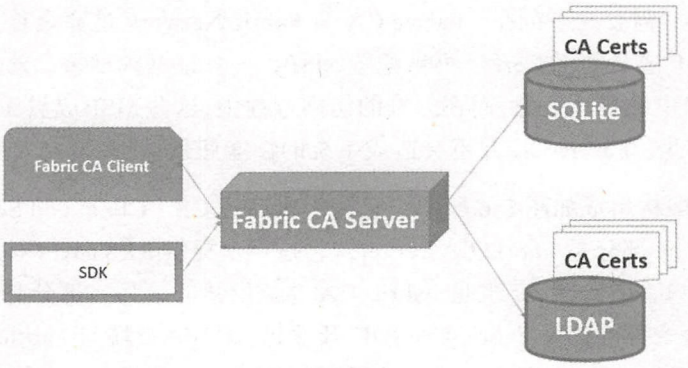


图6-7

Fabric CA Server 可以实现高可用的集群模式，因为 Fabric CA Server 提供的是 REST 服务接口，因此可以很方便地使用类似 HA-Proxy 的负载均衡代理软件实现对集群方式的部署，集群中的所有 Fabric CA Server 都共享相同的数据库，如图 6-8 所示。

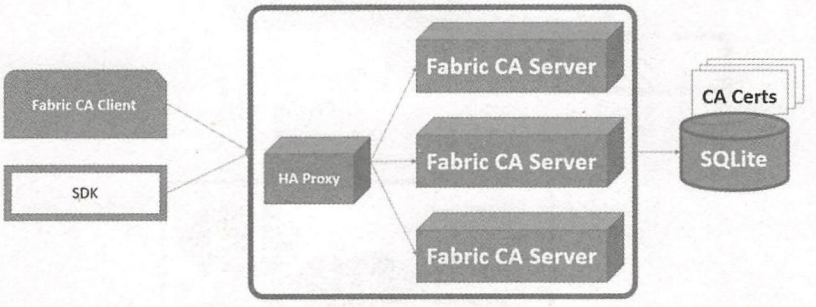


图6-8

为了安全起见，很多 CA 机构都采用了根证书(Root CA)+中间证书(Intermediate CA) 的根证书链的实现方案。在通常情况下，Root CA 不会直接为客户端应用签证，它们会先为自己生成几个中间证书，这几个中间证书作为 Root CA 的代表为客户端应用签证，中间证书还可以产生下一级中间证书，多级中间证书可以减少根证书的管理负担。此外，大部分 CA 方案并不会提供证书吊销机制（CRL），一旦中间证书的私钥泄漏或者证书过期，则可以直接吊销中间证书并给用户颁发新的证书。切记，一定要在绝对安全的环境下创建 Root CA 证书，确保私钥不会被泄露，可以采用禁用网络、拔掉网线的极端做法。下面是 Fabric 官方给出的增加了 HA Proxy 及根证书服务器与中间证书服务器的 Fabric CA 集群架构示意图，如图 6-9 所示。

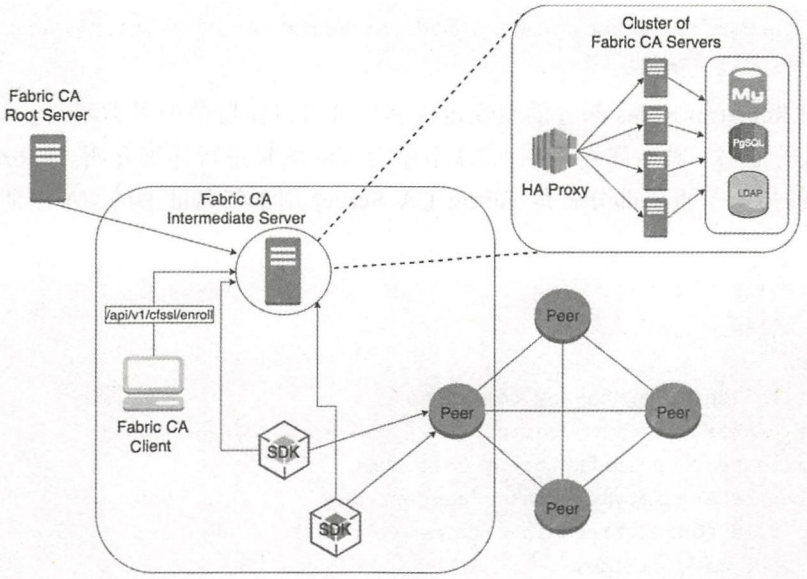


图6-9

6.3.2　Fabric CA 安装及功能

Fabric CA 提供了两种运行方式：传统的命令行方式及 Docker 容器方式，这里以 Docker 容器方式为例进行说明。在访问 Fabric CA Server 时需要一个授权用户，这通常是一个拥有管理权限的用户，因此在 Fabric CA Server 的配置文件 fabric-ca-server-config.yaml 中有下面一段内容来定义这个初始管理员 admin 的信息：

```
identities:
    - name: admin
      pass: adminpw
      type: client
      affiliation: ""
      attrs:
        hf.Registrar.Roles: "client,user,peer,orderer,validator,auditor,ca"
        hf.Registrar.DelegateRoles: "client,user,validator,auditor"
        hf.Revoker: true
        hf.IntermediateCA: true
        hf.AffiliationMgr: true
        hf.GenCRL: true
        hf.Registrar.Attributes:
```

```
"hf.AffiliationMgr,hf.Registrar.Attributes,hf.Registrar.Roles,hf.Revoker,customName
,hf.Registrar.DelegateRoles"
```

其中，hf.Registrar.Roles 部分的属性定义客户端可以注册的证书类型，比如用于 Peer、Orderer 节点的证书。其他属性将在 6.3.3 节结合实际场景进行详细介绍。Fabric CA Server 的配置文件的另一个重要部分就是 Fabric CA Server 用于签名证书的 CA 根证书的配置信息：

```
ca:
  # Name of this CA
  name: CA
  # Key file (default: ca-key.pem)
  keyfile: /shared/crypto-config/ca/key.pem
  # Certificate file (default: ca-cert.pem)
  certfile: /shared/crypto-config/ca-cert.pem
  # Chain file (default: chain-cert.pem)
  #chainfile: ca-chain.pem
  chainfile: /shared/crypto-config/ca-chain.pem
```

这里的 chainfile 属性定义了被 CA 根证书所信任的证书，如果我们配置了 chainfile，则它与 CA 根证书（Certfile）一起形成证书链，后者是证书链上的第 1 个证书（根节点证书）。

接下来是对 Fabric CA Server 的配置文件中 CSR（证书签名请求）部分的定义，如果我们在生成 CSR 的时候没有提供相应的属性，则 Fabric CA Server 会采用以下默认值：

```
csr:
   cn: ca.example.com
   names:
      - C: US
        ST: "North Carolina"
        L:
        O: Hyperledger
        OU: Fabric
   hosts:
    - 192.168.18.134
   ca:
      pathlen:
      pathlenzero:
      expiry:
```

接下来是对证书签名时的相关默认属性的定义，这里最重要的属性是证书的有效期：

```
signing:
   profiles:
     ca:
        usage:
          - cert sign
        expiry: 8000h
        caconstraint:
          isca: true
   default:
     usage:
       - cert sign
     expiry: 8000h
```

Fabric CA Server 生成的证书采用了椭圆曲线加密算法，我们可以通过下面的配置选项指定该算法的一些参数，比如密码强度：

```
bccsp:
   default: SW
   sw:
      hash: SHA2
      security: 256
      filekeystore:
         # The directory used for the software file-based keystore
#          keystore: /opt/keystore
         keystore: /shared/crypto-config/
```

如果我们要声明 Org 之间的从属关系，则可以类似这样定义：

```
affiliations:
  org1:
    - department1
    - department2
  org2:
    - department1
```

Fabric CA Server 生成的证书（确切地说应该是证书里的属性数据）可以被存放在数据库如 SQLite 3、MySQL 或者 PostgreSQL 中，默认存放在 SQLite 3 中。ca.yaml 中的配置如下：

```
db:
  type: sqlite3
  datasource: /opt/fabric-ca-server.db
```

假如 Fabric CA Server 的配置文件 fabric-ca-server-config.yaml 位于/hyperledger/data/shared/cas 目录下，则我们可以通过 Docker Volume 方式将其映射到容器内部，并且用环境变量 CONFIGYAML 来指定 ca.yaml 的路径。完整的 Docker 启动命令如下：

```
docker run \
--name fabric-ca \
--restart=always \
-p 7054:7054 \
-e CONFIGYAML=/shared/cas/ca.yaml \
-v /hyperledger/data/shared:/shared \
hyperledger/fabric-ca:x86_64-1.1.0 \
sh -c 'fabric-ca-server start -c ${CONFIGYAML} \
--cfg.affiliations.allowremove \
--cfg.identities.allowremove'
```

在 Docker 的启动命令中，为了演示删除从属关系和删除身份信息的功能，添加了--cfg.affiliations.allowremove 和--cfg.identities.allowremove 两个参数，默认是禁止删除从属关系和身份信息的。

我们使用 docker exec -it fabric-ca bash 命令进入容器，并设置 Fabric CA Client 的工作目录。Fabric CA Client 端的工作目录由以下几个条件决定，按照优先级排列，分别是：

◎ FABRIC_CA_CLIENT_HOME 环境变量；
◎ FABRIC_CA_HOME 环境变量；
◎ CA_CFG_PATH 环境变量；
◎ $ HOME/.fabric-ca-client 目录。

因为在 Server 端默认使用 FABRIC_CA_HOME 环境变量，所以我们使用 FABRIC_CA_CLIENT_HOME 环境变量配置 Client 端的工作目录：

```
export FABRIC_CA_CLIENT_HOME=/etc/hyperledger/fabric-ca-client
```

在操作 Fabric CA 的所有功能时都需要有一个身份认证证书。在介绍 Fabric CA Server 端的配置文件时，已经定义了一个初始的管理员身份。在这里我们登记这个管理员的身份认证证书，命令如下：

```
fabric-ca-client enroll -u http://admin:adminpw@localhost:7054
```

在执行以上命令后，可以看到以下日志：

```
[INFO] Created a default configuration file at
/etc/hyperledger/fabric-ca-client/fabric-ca-client-config.yaml
```

```
    [INFO] generating key: &{A:ecdsa S:256}
    [INFO] encoded CSR
   [INFO] Stored client certificate at
/etc/hyperledger/fabric-ca-client/msp/signcerts/cert.pem
    [INFO] Stored root CA certificate at
/etc/hyperledger/fabric-ca-client/msp/cacerts/localhost-7054.pem
```

我们通过日志可以发现，身份认证证书已经被保存在 Client 的工作目录下。接下来介绍 Fabric CA 的功能，并演示命令行的用法。

1. 从属关系管理

参与交易的多个交易主体可能属于不同的联盟，也可能属于同一个联盟的不同组织，还可能属于同一个组织的不同部门等。Fabric CA 提供了维护这种从属关系的功能。

1）查看从属关系

执行命令 fabric-ca-client affiliation list，可以看到当前 Fabric CA 默认注册的从属关系。默认注册的从属关系是在 Fabric CA Server 的配置文件中定义的，在本章的前半部分有相关介绍：

```
   affiliation: .
     affiliation: org2
       affiliation: org2.department1
     affiliation: org1
       affiliation: org1.department1
       affiliation: org1.department2
```

2）添加从属关系

添加一个 Org3 的从属关系：

```
   fabric-ca-client affiliation add org3

   [INFO] Configuration file location:
/etc/hyperledger/fabric-ca-client/fabric-ca-client-config.yaml
   Successfully added affiliation: org3
```

3）更新从属关系

修改刚才添加的从属关系 Org3 的名称为 org3-1：

```
   fabric-ca-client affiliation modify org3 --name org3-1
```

```
   [INFO] Configuration file location:
/etc/hyperledger/fabric-ca-client/fabric-ca-client-config.yaml
   Successfully modified affiliation: &{AffiliationInfo:{Name:org3-1 Affiliations:[]
Identities:[]} CAName:CA}
```

4）删除从属关系

删除刚才添加的从属关系 org3-1，在默认情况下是不允许删除从属关系的。

在启动 Fabric CA Server 时，在添加--cfg.affiliations.allowremove 后才能删除从属关系：

```
   fabric-ca-client affiliation remove org3-1

   [INFO] Configuration file location:
/etc/hyperledger/fabric-ca-client/fabric-ca-client-config.yaml
   Successfully removed affiliation: &{AffiliationInfo:{Name:org3-1 Affiliations:[]
Identities:[]} CAName:CA}
```

2. 用户身份管理

1）查看用户身份

代码如下：

```
   fabric-ca-client identity list
```

可以看到有一个 admin 的用户身份，它就是初始管理员的身份：

```
   Name: admin, Type: client, Affiliation: , Max Enrollments: -1, Attributes:
   [{
    Name: hf.Revoker Value: 1 ECert: false
   } {
    Name: hf.IntermediateCA Value: 1 ECert: false
   } {
    Name: hf.AffiliationMgr Value: 1 ECert: false
   } {
    Name: hf.GenCRL Value: 1 ECert: false
   } {
    Name: hf.Registrar.Attributes Value: hf.AffiliationMgr,
    hf.Registrar.Attributes,
    hf.Registrar.Roles,
    hf.Revoker,
    customName,
```

```
    hf.Registrar.DelegateRoles ECert: false
   } {
    Name: hf.Registrar.Roles Value: client,
    user,
    peer,
    orderer,
    validator,
    auditor,
    ca ECert: false
   } {
    Name: hf.Registrar.DelegateRoles Value: client,
    user,
    validator,
    auditor ECert: false
   }]
```

2）添加用户身份

有两种方式可以添加用户身份。

第 1 种是 JSON 格式，命令如下：

```
   fabric-ca-client identity add user1 --json '{"secret": "user1pw", "type": "user",
"affiliation": "org1", "max_enrollments": 1, "attrs": [{"name": "hf.Revoker", "value":
"true"}]}'

   [INFO] Configuration file location:
/etc/hyperledger/fabric-ca-client/fabric-ca-client-config.yaml
   Successfully added identity - Name: user1, Type: user, Affiliation: org1, Max
Enrollments: 1, Secret: user1pw, Attributes: [{Name:hf.Type Value:user ECert:true}
{Name:hf.Affiliation Value:org1 ECert:true} {Name:hf.Revoker Value:true ECert:false}
{Name:hf.EnrollmentID Value:user1 ECert:true}]
```

第 2 种是指定参数的方式，命令如下：

```
   fabric-ca-client identity add user2 --secret user2pw --type user --affiliation .
--maxenrollments 1 --attrs hf.Revoker=true

   [INFO] Configuration file location:
/etc/hyperledger/fabric-ca-client/fabric-ca-client-config.yaml
   Successfully added identity - Name: user2, Type: user, Affiliation: , Max Enrollments:
1, Secret: user2pw, Attributes: [{Name:hf.Revoker Value:true ECert:false}
{Name:hf.EnrollmentID Value:user2 ECert:true} {Name:hf.Type Value:user ECert:true}
{Name:hf.Affiliation Value: ECert:true}]
```

3）修改用户身份

同样有 JSON 和参数这两种方式可以修改用户身份。

JSON 方式的命令如下：

```
   fabric-ca-client identity modify user1 --json '{"secret": "newPassword",
"affiliation": ".", "attrs": [{"name": "hf.Registrar.Roles", "value": "peer,client"}]}'

   [INFO] Configuration file location:
/etc/hyperledger/fabric-ca-client/fabric-ca-client-config.yaml
   Successfully modified identity - Name: user1, Type: user, Affiliation: , Max
Enrollments: 1, Secret: newPassword, Attributes: [{Name:hf.Registrar.Roles
Value:peer,client ECert:false} {Name:hf.Revoker Value:true ECert:false}
{Name:hf.EnrollmentID Value:user1 ECert:true} {Name:hf.Type Value:user ECert:true}]
```

参数方式的命令如下：

```
   fabric-ca-client identity modify user1 --secret newsecret

   [INFO] Configuration file location:
/etc/hyperledger/fabric-ca-client/fabric-ca-client-config.yaml
   Successfully modified identity - Name: user1, Type: user, Affiliation: , Max
Enrollments: 1, Secret: newsecret, Attributes: [{Name:hf.Registrar.Roles
Value:peer,client ECert:false} {Name:hf.Revoker Value:true ECert:false}
{Name:hf.EnrollmentID Value:user1 ECert:true} {Name:hf.Type Value:user ECert:true}]
```

4）删除用户身份

默认是不允许删除用户身份的，可以通过添加 Fabric CA Server 的启动参数来删除：

```
   --cfg.identities.allowremove来开启该功能:
   fabric-ca-client identity remove user1

   [INFO] Configuration file location:
/etc/hyperledger/fabric-ca-client/fabric-ca-client-config.yaml
   Successfully removed identity - Name: user1, Type: user, Affiliation: , Max Enrollments:
1, Attributes: [{Name:hf.Revoker Value:true ECert:false} {Name:hf.EnrollmentID
Value:user1 ECert:true} {Name:hf.Type Value:user ECert:true} {Name:hf.Registrar.Roles
Value:peer,client ECert:false}]
```

3. 身份证书管理

1）登记身份证书

在本节前半部分已经登记了初始管理员的身份证书。在这里将登记 user2 用户的身份证书：

```
   fabric-ca-client enroll -u http://user2: user2pw@localhost:7054

   [INFO] generating key: &{A:ecdsa S:256}
   [INFO] encoded CSR
   [INFO] Stored client certificate at
/etc/hyperledger/fabric-ca-client/msp/signcerts/cert.pem
   [INFO] Stored root CA certificate at
/etc/hyperledger/fabric-ca-client/msp/cacerts/localhost-7054.pem
```

2）重新登记身份证书

代码如下：

```
   fabric-ca-client reenroll -u http://user2: user2pw@localhost:7054

   [INFO] Configuration file location:
/etc/hyperledger/fabric-ca-client/fabric-ca-client-config.yaml
   [INFO] generating key: &{A:ecdsa S:256}
   [INFO] encoded CSR
   [INFO] Stored client certificate at
/etc/hyperledger/fabric-ca-client/msp/signcerts/cert.pem
   [INFO] Stored root CA certificate at
/etc/hyperledger/fabric-ca-client/msp/cacerts/localhost-7054.pem
```

3）撤销身份证书

代码如下：

```
   fabric-ca-client revoke -e user2 -r 1

   [INFO] Configuration file location:
/etc/hyperledger/fabric-ca-client/fabric-ca-client-config.yaml
   [INFO] Sucessfully revoked certificates:
[{Serial:17d1b78f2aeded0980fd4f9bfd5f8e273f2f7752
AKI:6bbd298cb1b0ca009f2414ac5ad50996ecf318ef}
{Serial:61c1f7d1f290b4994d5252ba5d8dc988fb5a67af
AKI:6bbd298cb1b0ca009f2414ac5ad50996ecf318ef}]
```

本节介绍了 Fabric CA Server 的安装及 Fabric CA Client 的命令行使用方式。在 6.3.3 节中除了会介绍通过 SDK 进行编程的内容，还会重点介绍从属关系管理、用户身份管理及身份证书管理的约束条件。

6.3.3 Fabric CA SDK 编程

1. 从属关系管理

1）前置条件

当前登记员（管理员，以下简称登记员）的身份必须满足以下所有条件，才能够更新从属关系。

- 当前登记员的身份必须具有属性 hf.AffiliationMgr 并且为 true。
- 当前登记员的从属关系必须高于要更新的从属关系。例如，如果当前登记员的从属关系是 ab，则可以添加从属关系 abc，而不是 a 或者 ab。

2）添加从属关系

添加从属关系的代码如下：

```
// 构建HFCAAffiliation对象
HFCAAffiliation affiliation = ca.newHFCAAffiliation("org1.department1.team1");
// 使用admin身份添加org1.department1.team1从属关系
affiliation.create(admin);
```

3）更新从属关系

如果更新的从属关系已经绑定了身份，则在默认情况下会报错。如果也要更新已经绑定的身份，则可以添加 force 选项：

```
// 构建HFCAAffiliation 对象，设置从属关系为 org1.department1
HFCAAffiliation affiliation = ca.newHFCAAffiliation("org1.department1");
// 设置要更新的从属关系为org1.department3
affiliation.setUpdateName("org1.department3");
// 使用admin身份更新从属关系
// 第2个参数true表示：同时更新已经绑定到org1.department1上的所有身份信息为org1.department3
affiliation.update(admin,true);
```

4）删除从属关系

在默认情况下，在 fabric-ca-server 中删除从属关系信息是被禁止的，但可以通过添加命令行参数的-cfg.affiliations.allowremove 选项启用删除用户身份信息的功能。

如果将要删除的从属关系已经绑定了身份，则在默认情况下会报错。如果要删除已经绑定的身份，则可以添加 force 选项进行：

```
// 构建HFCAAffiliation 对象，设置从属关系为 org1.department2
HFCAAffiliation affiliation = ca.newHFCAAffiliation("org1.department2");
// 使用admin身份，删除从属关系
// 第2个参数true表示：同步删除已经绑定org1.department2的所有身份
affiliation.delete(admin,true);
```

5）查询从属关系

查询从属关系的代码如下：

```
// 获取某个特定的从属关系
HFCAAffiliation affiliation = ca.newHFCAAffiliation("org1");
affiliation.read(admin);
affiliation.getChildren()
.stream().map(f->f.getName()).forEach(System.out::println);
// 获取当前admin身份有权查看的所有从属关系
ca.getHFCAAffiliations(admin)
.getChildren()
.stream().map(f->f.getName()).forEach(System.out::println);
```

2. 用户身份管理

1）前置条件

在进行动态更新身份时，需要同时满足以下两个条件，否则授权失败。

- 客户端身份必须具有 hf.Registrar.Roles 属性，不仅是使用逗号分隔的值列表，而且只能是值列表中的一个值。例如，在更新前 hf.Registrar.Roles=peer,client，那么只能更新身份类型为 Peer 或者 Client，而不能是 Orderer。
- 客户端身份的从属关系必须是等于或者匹配更新前从属关系的前缀。例如，如果 hf.Affiliation=org1，那么可以更新为 hf.Affiliation=org1111 或者 hf.Affiliation =org1.department1，即新的值必须以 org1 开头。

2）查询身份信息

可以获取到满足条件的所有身份信息：

```
        // user代表客户端身份
        ca.getHFCAIdentities(user)
        .stream()
        .map(f->f.getEnrollmentId())
        .forEach(System.out::println); // 遍历获取的身份列表
```

查询某个身份的信息方法如下：

```
        // 设置要获取的用户身份信息
        HFCAIdentity identity  = ca.newHFCAIdentity(newUserName);
        // 以admin身份读取newUserName的用户信息
        System.out.println(identity.read(admin));
```

3）添加身份信息

通过 HFCAIdentity 对象注册用户身份，并没有自动生成身份认证证书：

```
      SampleUser admin = sampleStore.getMember(adminName, orgName);
    if (!admin.isEnrolled()) {
     admin.setEnrollment(ca.enroll(admin.getName(),adminPassword ));
     admin.setMspId(mspId);
    }
    // 构建HFCAIdentity 对象，并通过HFCAIdentity对象添加用户身份
    // 这里只是添加了用户身份，并没有生成用户身份认证证书（ECert）
    HFCAIdentity identity = ca.newHFCAIdentity(newUserName);
    SampleUser user = sampleStore.getMember(newUserName,orgName);
    if (!user.isRegistered()) {
    // 设置用户从属关系
     identity.setAffiliation("org1.department1");
    // 设置最大拉取次数
     identity.setMaxEnrollments(-1);
    // 设置密码
     identity.setSecret(newUserPassword);
    // 设置用户类型，该类型的约束条件已经在前面的章节中详细介绍过，这里不再赘述
     identity.setType("user");
    // 设置用户属性
     List<Attribute> attrsList = new ArrayList<>();
     Attribute attrs = new
Attribute("hf.Registrar.Attributes","hf.Registrar.Attributes,hf.Revoker,hf.Registra
```

```
r.Roles");
    attrsList.add(attrs);
    Attribute revoker = new Attribute("hf.Revoker","true");
    attrsList.add(revoker);
    Attribute roles = new Attribute("hf.Registrar.Roles","user,client,peer");
    attrsList.add(roles);
   // 设置用户自定义的属性
    Attribute customAttr = new Attribute("customName","customValue",true);
    attrsList.add(customAttr);
    identity.setAttributes(attrsList);
   // 注册用户信息
    identity.create(admin);
  }
```

表 6-1 列出了身份的所有字段及它们是必填的还是可选的，以及默认值。

表 6-1

字　段	必填还是可选	默 认 值
ID	必填	
Secret	可选	
Affiliation	可选	登记员的 Affiliation
Type	可选	Client
Maxenrollments	可选	0
Attributes	可选	

4）更新身份信息

动态更新身份信息需要构建新的 HFCAIdentity 对象。可以一次进行多项修改，任何未修改的元素都将保留其原始值。

需要注意的是，maxenrollments 如果被设置为-2，则表示 Fabric CA 的最大登记次数。

下面是一个同时改变身份类型和最大登记次数的例子：

```
HFCAIdentity identity = ca.newHFCAIdentity(newUserName);
 identity.setType("client");
 identity.setMaxEnrollments(100);
```

```
identity.update(admin);
```

5）删除身份信息

在默认情况下，在 fabric-ca-server 中删除用户身份信息是被禁止的，但可以通过添加命令行参数的-cfg.identities.allowremove 选项来启用删除用户身份信息的功能。

下面是删除 user1 用户身份并且撤销与该身份关联的所有认证证书的例子：

```
// 构建要删除的用户身份信息对象。deleteUserName为用户的身份ID
HFCAIdentity identity = ca.newHFCAIdentity(deleteUserName);
// 使用admin登记员身份执行删除操作
identity.delete(admin);
```

3. 身份证书管理

1）注册并登记身份证书

接下来演示注册新用户的步骤。

（1）构建 SampleStore 对象，我们获取到的身份认证证书会被保存在这里。

（2）构建 HFClient 的实例，设置 CryptoSuite。

（3）获取并设置组织管理员的信息，如果不存在，就先通过 enroll 获取组织管理员的注册证书和私钥信息。

（4）根据新注册用户的信息构建 RegistrationRequest 请求，并调用 HFCAClient 的 register 接口进行注册。HFCAClient 会调用 Fabric CA 的 RESTApi 注册用户信息。

（5）新用户在注册完成后，会调用 HFCAClient 的 enroll 接口拉取身份认证证书和私钥。

以下为示例代码：

```
String orgName = "Org1MSP";
String mspId   = "Org1MSP";
String caLocation = "http://192.168.18.134:1149";
String caName = "CA";
String adminName = "new24-user";
String adminPassword = "123456";
String newUserName = "new26-user";
String newUserPassword = "123456";
```


```
    SampleOrg  sampleOrg = new SampleOrg(orgName,mspId);
    String kstore = "f:/fabricstore";
    SampleStore sampleStore = new SampleStore(new File(kstore));
    HFCAClient ca = HFCAClient.createNewInstance(caName,caLocation,null);
    ca.setCryptoSuite(CryptoSuite.Factory.getCryptoSuite());
    HFCAInfo info = ca.info();
    assertNotNull(info);
    String infoName = info.getCAName();
    if (infoName != null && !infoName.isEmpty()) {
     assertEquals(ca.getCAName(), infoName);
    }
    SampleUser admin = sampleStore.getMember(adminName, orgName);
    if (!admin.isEnrolled()) {
     admin.setEnrollment(ca.enroll(admin.getName(),adminPassword ));
     admin.setMspId(mspId);
    }

    // 根据配置信息构建需要注册的新用户
    SampleUser user = sampleStore.getMember(newUserName,orgName);
    if (!user.isRegistered()) {
    RegistrationRequest rr = new
RegistrationRequest(user.getName(),"org1.department1");
      rr.setSecret(newUserPassword);
      // 添加hf.Revoker属性，该用户的身份认证证书允许撤销
      Attribute revoker = new Attribute("hf.Revoker","true");
      rr.addAttribute(revoker);
    // 添加hf.Registrar.Roles属性，如果添加该属性，则可以使用新注册的用户身份来添加新的用户类型
    Attribute roles = new Attribute("hf.Registrar.Roles","user,client,peer");
      rr.addAttribute(roles);
```

下面设置 hf.Registrar.Attributes 属性，这里举个例子说明 hf.Registrar.Attributes 属性的作用。假设当前用户是管理员 A，管理员 A 在创建用户 B 时，赋予了用户 B 创建其他用户的功能，同时指定了用户 B 在创建其他用户时可以设置的属性列表，不在这个列表中的属性不允许用户 B 为其他用户添加：

```
    Attribute attrs = new
Attribute("hf.Registrar.Attributes","hf.Revoker,hf.Registrar.Roles");
      rr.addAttribute(attrs);
      user.setEnrollmentSecret(ca.register(rr, admin));
    }
    // 在注册完成后拉取用户证书
```


```
if (!user.isEnrolled()) {
 user.setEnrollment(ca.enroll(user.getName(), user.getEnrollmentSecret()));
 user.setMspId(mspId);
}
sampleOrg.addUser(user);
```

上面是一个注册默认的用户身份的例子，而在实际生产环境下，往往需要有适当权限的用户。接下来介绍在使用注册用户功能前需要满足哪些条件。

（1）注册的新用户类型，只能是在当前登记员属性 hf.Registrar.Roles 中包含的角色。例如，如果当前登记员的 hf.Registrar.Roles 包含 peer、app 和 user 三个角色，那么注册新用户的用户角色只能是 peer、app 和 user 类型的身份，但不能注册 orderer 角色的身份：

```
RegistrationRequest rr = new RegistrationRequest(user.getName());
rr.setType("peer");   // 这里是不允许注册 orderer 角色的用户身份的
```

如果没有设置，则默认为 client 角色。

（2）注册的新用户从属关系必须从属于当前登记员的从属关系。例如，如果当前登记员从属于 org1.department1，那么新注册的用户一定从属于 org1.department1 及其下属。如果新用户有其他从属关系，则是不允许其注册的。例如：

```
new RegistrationRequest(user.getName(),"org1.department2");
// 这里 org1.department2并不从属于org1.department1，是不允许其注册的
```

（3）以 hf 开头的属性是 Fabric CA 的默认属性，只有当前登记员拥有该属性并且该属性是 hf.Registrar.Attributes 属性的一部分时，才可以注册。例如，如果想让新用户的证书也可以被撤销，即添加 hf.Revoker 属性，那么当前登记员也必须拥有 hf.Revoker 属性，并且在当前登记员的 hf.Registrar.Attributes 属性列表中必须包含 hf.Revoker，即 hf.Registrar.Attributes="xxxx, hf.Revoker"。

（4）如果注册自定义属性，则登记员必须拥有 hf.Registrar.Attributes 属性或模式的值。当前模式只支持尾部带星号的字符串。例如 hf.Registrar.Attributes=ab*指匹配所有以 ab 开头的属性名。

（5）如果请求的属性名称是 hf.Registrar.Attributes，则执行额外的检查，查看该属性的请求值是否为登记员的属性值的子集。

进行身份注册的所有属性（属性名称区分大小写）列表如表 6-2 所示。

表 6-2

名　称	类　型	描　述
hf.Registrar.Roles	数组	允许登记员注册的角色列表
hf.Registrar.DelegateRoles	数组	允许登记员在注册新用户时赋予新用户的角色列表
hf.Registrar.Attributes	数组	登记员可以注册的属性列表
hf.GenCRL	Boolean 类型	为 true 时，可以生成证书撤销列表
hf.Revoker	Boolean 类型	为 true 时，可以撤销用户或证书
hf.AffiliationMgr	Boolean 类型	为 true 时，可以管理从属关系
hf.IntermediateCA	Boolean 类型	为 true 时，可以作为一个中级 CA

2）重新登记身份证书

在 3.1.1 节只演示了注册用户和获取身份认证证书的过程。如果身份证书过期，就需要重新获取新的身份认证证书。可以使用 ca.reenroll 接口重新拉取新的身份认证证书：

```
ca.reenroll(user);
```

3）撤销身份或身份认证证书

只有在用户身份信息中包含 hf.Revoker 属性并且值为 true，还包含 hf.Registrar.Roles 属性时，才可以撤销身份或身份认证证书。

- 撤销身份将撤销身份所拥有的所有证书，并且会阻止使用已撤销的身份获取任何新的证书。
- 撤销证书将使单个证书无效。
- 撤销证书的角色类型只能是在登记员自身的 hf.Registrar.Roles 属性中包含的角色类型。
- 只能对与登记员自身的从属关系相同或前缀相同的身份或者证书执行撤销操作。

需要注意的是，在 fabric-sdk-java 1.1.0 版本中，如果 user 的 serial 和 aki 有值，则执行的是撤销单个身份认证证书的操作。如果 user 的 serial 和 aki 至少有一个为空，则执行的是撤销身份的操作。但是当前版本的 API 不能执行撤销身份的操作。如果需要执行撤销身份的操作，则可以自行对 API 进行扩展。

下面通过一个例子来说明。在讲解用户身份管理时所举的例子中，我们在注册用户时，给用户信息附加了 hf.Revoker 属性，然后进行注册。可以将相应的代码改造为：

```
    RegistrationRequest rr = new RegistrationRequest(user.getName(),
"org1.department1");
    // 附加hf.Revoker=true 属性
    Attribute attr = new Attribute("hf.Revoker","true");
    rr.addAttribute(attr);
    user.setEnrollmentSecret(ca.register(rr, admin));
    // 在注册完成后拉取用户证书
    if (!user.isEnrolled()) {
    user.setEnrollment(ca.enroll(user.getName(), user.getEnrollmentSecret()));
            user.setMspId(mspId);
    }
    // 撤销用户证书，只有包含hf.Revoker=true属性的用户身份证书可以被撤销
    ca.revoke(user,user.getEnrollment()," revoke test");
```

4）基于属性的访问控制

访问控制决策可以通过智能合约根据身份认证证书的属性进行，这也被称为基于属性的访问控制（ABAC）。为了实现这一点，身份认证证书可能包含一个或者多个属性名称和值，智能合约在提取属性值后做访问控制。

例如，假设正在开发一个应用程序 app1，并且希望某个智能合约的操作只能由 app1 的管理员访问，则这时在智能合约内可以通过验证身份认证证书中是否包含名为 app1Admin 的值为 true 的属性，来判断当前用户身份是否是 app1 的管理员。当然，app1Admin 是我们自定义的属性名称，属性值也可以是任意值，不一定是 Boolean 类型。

有两种方法可以获取具有属性的身份认证证书，如下所述。

（1）在注册身份时，可以指定为身份颁发认证证书时默认包含的属性。默认属性可以在注册身份认证证书时被覆盖，但是这对于建立行为非常有用，如果注册身份认证证书是在应用程序之外进行的，就不需要再修改应用程序了。

在讲解从属关系管理所举的例子中，我们使用了 Attribute 对象，它的构造方法还有第 3 个参数，该参数是 Boolean 类型的。如果我们将其设置为 true，就表示相对的属性值会被插入用户身份认证证书中，可以用它来做访问控制。例如，下面这行代码的第 3 个参数表示将 hf.Revoker 添加到身份认证证书中：

```
    Attribute revoker = new Attribute("hf.Revoker","true",true);
```


（2）在登记身份获取身份认证证书时，可以显式添加一个或多个属性到证书中。对于每个请求的属性，可以指定该属性是否要被添加到身份认证证书中，如果没有指定并且身份也不具有该属性，就会发生错误。

需要注意的是，在显式指定要添加到证书中的属性时有一个配置项，可以配置要添加的属性不在证书属性列表中时是否报错的选项。代码如下：

```
// 构建EnrollmentRequest对象
EnrollmentRequest enrollmentRequest = new EnrollmentRequest();
// 显式指定要添加到证书中的属性，如果不存在就报错
enrollmentRequest.addAttrReq("hf.Revoker");
// 显式指定要添加到证书中的属性，如果不存在就报错
enrollmentRequest.addAttrReq("hf.Affiliation");
// 显式指定要添加到证书中的属性，如果不存在则不报错
enrollmentRequest.addAttrReq("notexistsinuser").setOptional(true);
user.setEnrollment(ca.enroll(user.getName(),
user.getEnrollmentSecret(),enrollmentRequest));
```

身份认证证书是 JSON 格式的，以下是包含 hf.Affiliation=org1.department1 和 hf.Revoker=true 的证书样本，可以看到在 JSON 格式中包含多个属性。非常重要的是，如果使用外部 CA 证书的属性，则必须在 X509 v3 extension 部分包含 1.2.3.4.5.6.7.8.1 的参数，并且值为一个包含身份属性的 JSON 字符串。

我们借助 OpenSSL 工具打印出该证书的内容：

```
        …证书的其他部分
            1.2.3.4.5.6.7.8.1:
                {"attrs":{"hf.Affiliation":"org1.department1","hf.Revoker":"true"}}
    Signature Algorithm: ecdsa-with-SHA256
        30:44:02:20:1e:fa:2d:d6:69:0b:9b:13:5d:0f:53:1d:f4:e8:
        8e:fc:5b:f2:cf:54:ff:8a:01:b1:f9:af:fb:2a:48:e4:1d:43:
        02:20:32:75:8a:fe:d7:83:14:41:d5:0f:62:74:46:d9:f5:c9:
        86:67:85:ef:fe:c3:b1:2c:d3:b5:7e:c1:20:1e:0b:a9
```

在智能合约中实现 ABAC 的关键代码如下：

```
……
"github.com/hyperledger/fabric/core/chaincode/lib/cid"
……
// 判断在证书中是否包含hf.Affiliation 属性，并且值为org1.department1
err := cid.AssertAttributeValue(stub, "hf.Affiliation", "org1.department1")
 if err != nil {
```

```
        return shim.Error(err.Error())
    }
```

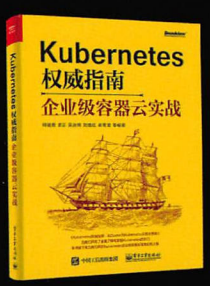
cid 提供的 API 列表如表 6-3 所示。

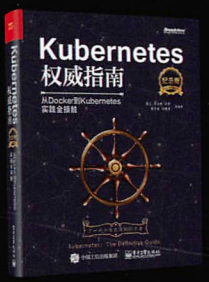
表 6-3

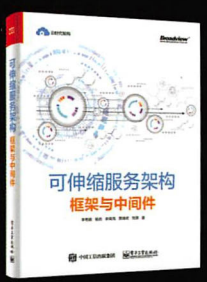
API	说　明
GetID	获取 client id
GetMSPID	获取 mspid
GetAttributeValue	获取属性值
AssertAttributeValue	验证在证书属性中是否有给定的 key 和 value 的属性
GetX509Certificate	获取 X509 证书

反侵权盗版声明

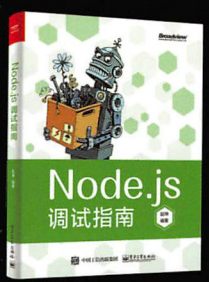
好书分享

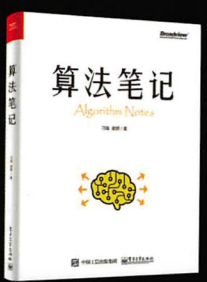

Kubernetes
权威指南
企业级容器云实战


Kubernetes
权威指南
从Docker到Kubernetes
实践全接触


可伸缩服务架构
框架与中间件


ARCHITECTURE
DECODING
架构解密
从分布式到微服务


Node.js
调试指南


算法笔记
Algorithm Notes


分布式数据库架构
及企业实践
基于Mycat中间件


分布式服务架构
原理、设计与实战

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

区块链轻松上手

原理、源码、搭建与应用

2016年，在解决客户的结算支付系统的架构选型问题时，我和治辉相识，当时我便深深折服于他对技术的专注及由此积累的深厚技术功底。这些年，从分布式数据库中间件、微服务架构到容器编排，治辉一直行走在技术前线，关于区块链的这次探索也不例外。关于区块链，众说纷纭，《区块链轻松上手：原理、源码、搭建与应用》从机制、机理的角度看区块链，能够帮助读者拨开迷雾，轻松掌握区块链的技术本质。

德勤管理咨询副总监 刘剑锋

区块链作为一项新兴技术，最早出现在以比特币作为数字加密货币的核心技术中，至今不过十年，但是价值不仅限于数字加密货币；更重要的是，它让各个组织、机构建立起天然的互信体系。《区块链轻松上手：原理、源码、搭建与应用》从区块链的基础知识开始讲起，以HyperLedger Fabric为主线，清晰地讲解了基于HyperLedger Fabric联盟链的搭建、开发、应用和解析等，能够帮助大家轻松上手区块链并掌握开发联盟链的基本技能。本书适合对区块链感兴趣的读者阅读和参考。

《可伸缩服务架构：框架与中间件》《分布式服务架构：原理、设计与实战》作者 李艳鹏

HyperLedger Fabric是非常优秀的区块链框架，实现了完备的权限控制和安全保障，可代表区块链技术的新高度。《区块链轻松上手：原理、源码、搭建与应用》是目前市面上对Fabric框架剖析更详尽、干货更多的一本书，它摒弃喧嚣，回归技术，深入浅出地讲解区块链的原理，并将理论和实践充分结合。我们在阅读本书时能充分感受到作者编著本书时的认真和严谨，在学习相关知识遇到问题时大多能在本书中找到答案。建议想系统学习区块链的同行都阅读本书。

拜腾汽车中国区软件研发总监 林海

欢迎本书读者和开源爱好者
加入区块链专家交流群：875130824

上架建议：区块链>HyperLedger Fabr

ISBN 978-7-121-34878-5
9 787121 348785 >
定价：79.00元

策划编辑：张国霞
责任编辑：孙学瑛
封面设计：李　玲

欢迎投稿
邮箱：zhanggx@phei.com.cn
微信：zgx228